Network Working Group
Request for Comments: 1037

B. Greenberg
S. Keene
December 1987

NFILE - A File Access Protocol

STATUS OF THIS MEMO

This document includes a specification of the NFILE file access protocol and its underlying levels of protocol, the Token List Transport Layer and Byte Stream with Mark. The goal of this specification is to promote discussion of the ideas described here, and to encourage designers of future file protocols to take advantage of these ideas. A secondary goal is to make the specification available to sites that might benefit from implementing NFILE. The distribution of this document is unlimited.

TABLE OF CONTENTS

		Page
1.	INTRODUCTION	3
2.	NFILE PROTOCOL LAYERING	4
3.	OVERVIEW OF AN NFILE SESSION	5
4.	NFILE CONTROL AND DATA CONNECTIONS	6
5.	NFILE FILE OPENING MODES	7
6.	NFILE CHARACTER SET	9
7.	CONVENTIONS USED IN THIS DOCUMENT	10
7.1	Mapping Data Types Into Token List Representation	10
7.2	Format of NFILE Commands and Responses	10
7.3	Data Channel Handles and Direct File Identifiers	13
7.4	Syntax of File and Directory Pathname Arguments	13
7.5	Format of NFILE File Property/Value Pairs	14
8.	NFILE COMMANDS	16
8.1	ABORT Command	16
8.2	CHANGE-PROPERTIES Command	16
8.3	CLOSE Command	17
8.4	COMPLETE Command	19
8.5	CONTINUE Command	20

8.6 CREATE-DIRECTORY Command 21
8.7 CREATE-LINK Command 21
8.8 DATA-CONNECTION Command 22
8.9 DELETE Command 23
8.10 DIRECT-OUTPUT Command 23
8.11 DIRECTORY Command 24
8.11.1 NFILE DIRECTORY Data Format 26
8.12 DISABLE-CAPABILITIES Command 27
8.13 ENABLE-CAPABILITIES Command 28
8.14 EXPUNGE Command 28
8.15 FILEPOS Command 29
8.15.1 Implementation Hint for FILEPOS Command 30
8.16 FINISH Command 30
8.17 HOME-DIRECTORY Command 31
8.18 LOGIN Command 32
8.19 MULTIPLE-FILE-PLISTS Command 34
8.20 OPEN Command 35
8.20.1 NFILE OPEN Optional Keyword/Value Pairs 39
8.20.2 NFILE OPEN Response Return Values 45
8.21 PROPERTIES Command 47
8.22 READ Command 49
8.23 RENAME Command 50
8.24 RESYNCHRONIZE-DATA-CHANNEL Command 51
8.24.1 Implementation Hints for RESYNCHRONIZE-DATA-CHANNEL Command 51
8.25 UNDATA-CONNECTION Command 52

9. NFILE RESYNCHRONIZATION PROCEDURE 53

9.1 NFILE Control Connection Resynchronization 54
9.2 NFILE Data Connection Resynchronization 55

10. NFILE ERRORS AND NOTIFICATIONS 58

10.1 Notifications From the NFILE Server 58
10.2 NFILE Command Response Errors 59
10.3 NFILE Asynchronous Errors 60
10.4 NFILE Three-letter Error Codes 61

11. TOKEN LIST TRANSPORT LAYER 65

11.1 Introduction to the Token List Transport Layer 65
11.2 Token List Stream 66
11.2.1 Types of Tokens and Token Lists 66
11.2.2 Token List Stream Example 68
11.2.3 Mapping of Lisp Objects to Token List Stream Representation 70
11.2.4 Aborting and the Token List Stream 71

11.3 Token List Data Stream 72

12. BYTE STREAM WITH MARK 73

12.1 Discussion of Byte Stream with Mark 73
12.2 Byte Stream with Mark Abortable States 75

13. POSSIBLE FUTURE EXTENSIONS 77

APPENDIX A. NORMAL TRANSLATION MODE 79

APPENDIX B. RAW TRANSLATION MODE 83

APPENDIX C. SUPER-IMAGE TRANSLATION MODE 84

NOTES 86

LIST OF TABLES

TABLE 1. TRANSLATIONS FROM NFILE CHARACTERS TO UNIX CHARACTERS 80
TABLE 2. TRANSLATIONS FROM UNIX CHARACTERS TO NFILE CHARACTERS 80
TABLE 3. TRANSLATIONS FROM NFILE TO PDP-10 CHARACTERS 81
TABLE 4. TRANSLATIONS FROM PDP-10 CHARACTERS TO NFILE CHARACTERS 82
TABLE 5. SUPER-IMAGE TRANSLATION FROM NFILE TO ASCII 84
TABLE 6. SUPER-IMAGE TRANSLATION FROM ASCII TO NFILE 85

1. INTRODUCTION

NFILE stands for "New File Protocol". NFILE was originally designed as a replacement for an older protocol named QFILE, with the goal of solving robustness problems of QFILE, hence the name "New File Protocol".

NFILE was designed and implemented at Symbolics by Bernard S. Greenberg. Mike McMahon made important contributions, especially in the design and implementation of the Byte Stream with Mark and Token List Transport layers. NFILE has been used successfully for file access between Symbolics computers since 1985. NFILE servers have been written for UNIX hosts as well. NFILE is intended for use by any type of file system, not just the native Symbolics file system.

NFILE is a file access protocol that supports a large set of operations on files and directories on remote systems, including:

- Reading and writing entire files
- Reading and writing selected portions of files
- Deleting and renaming files

- Creating links
- Listing, creating, and expunging directories
- Listing and changing the properties of files
- Enabling and disabling access capabilities on a remote host

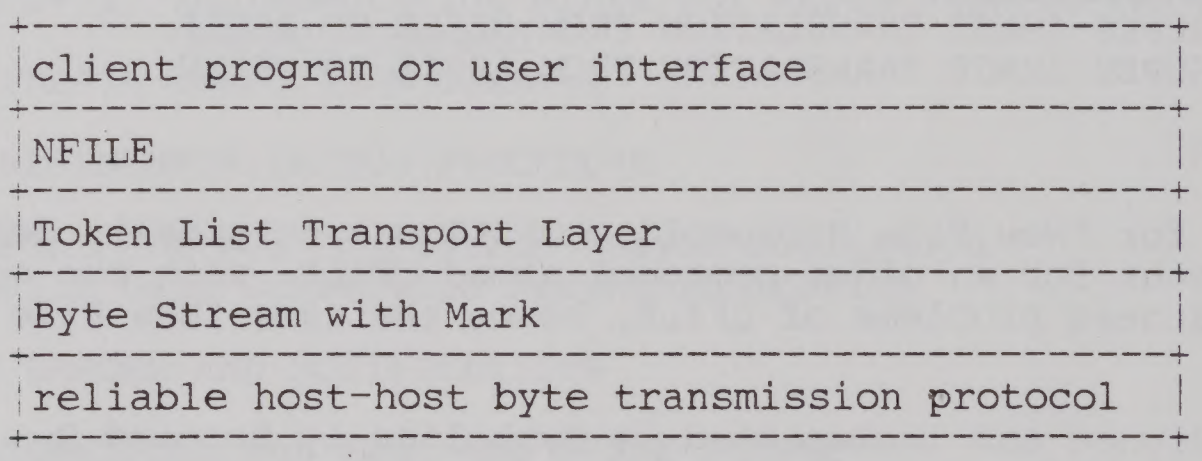
NFILE supports file transfer of binary or character files.

The NFILE server provides information about any errors that occur in the course of a command. In addition, NFILE has a robust scheme for handling aborts on the user side.

This specification defines NFILE user version 2 and server version 2. We do not envision NFILE as an unchanging protocol, but rather as a protocol that could continue to develop if additional requirements are identified though the process of this Request for Comments. The design of NFILE makes room for various useful extensions. Some of the extensions that we are considering are described later on in this document: See the section "Possible Future Extensions", section 13.

2. NFILE PROTOCOL LAYERING

NFILE is a layered file protocol. The layers are:



Byte Stream with Mark is a simple protocol that guarantees that an out-of-band signal can be transmitted in the case of program interruption. Byte Stream with Mark is to be layered upon extant byte stream protocols. An important goal of the NFILE design was that NFILE could be built on any byte stream. It is currently implemented on TCP and Chaosnet. See the section "Byte Stream with Mark", section 12.

The Token List Transport Layer is a protocol that facilitates the transmission of simple structured data, such as lists. See the section "Token List Transport Layer", section 11.

The NFILE commands and command responses are transmitted in "token lists". See the section "NFILE Commands", section 8.

This specification does not include a client program or user interface to the protocol. In the Symbolics implementation, the normal file operations transparently invoke NFILE, when the remote host is known to support NFILE. Another possible interface to NFILE would be through a dedicated client program that would issue NFILE commands in response to explicit requests by the user.

3. OVERVIEW OF AN NFILE SESSION

An NFILE session is a dialogue between two hosts. The host that initiates the NFILE session is known as the "user side", and the other host is the "server side". The user side sends all NFILE commands. The server receives each command, processes it, and responds to it, indicating the success or failure of the command.

The user side keeps track of commands sent and command responses received by using "transaction identifiers" to identify each command. The user side generates a transaction identifier (which must be unique per this dialogue) for each command, and sends the transaction identifier to the server along with the command. Each NFILE server response includes the transaction identifier of the command with which the response is associated. The server is not required to respond to commands in the same order that the user gave them.

The user side sends NFILE commands over a bidirectional network connection called the "control connection". The server sends its command responses on the same control connection. The control connection governing the NFILE session is established at the beginning of the session. If the control connection is ever broken, the NFILE session is ended.

Whereas NFILE commands and responses are transmitted on the control connection, file data is transferred over "data channels". An "input data channel" transfers data from server to user. An "output data channel" transfers data from user to server. Each input data channel is associated with an output data channel; together these two channels comprise a "data connection".

Often more than one NFILE activity is occurring at any given time. For example, several files can be open and transferring data simultaneously; one or more commands can be in process at the same time; and the server can be simultaneously transmitting directory lists and processing further commands. This happens in an implementation in which the user side has multiple processes, and several processes share a single NFILE server.

4. NFILE CONTROL AND DATA CONNECTIONS

The user and server communicate through a single control connection and a set of data connections. Data connections are established and disestablished by NFILE commands. The user side sends NFILE commands to the server over the control connection. The server responds to every user command over this control connection. The actual file data is transmitted over the data connections.

User aborts can disrupt the normal flow of data on the control connection and data connections. An important aspect of any file protocol is the way it handles user aborts. NFILE uses a resynchronization procedure to bring the affected control connection or data channel from an unknown, unsafe state into a known state. After resynchronization, the control connection or data channel can be reused. See the section "NFILE Resynchronization Procedure", section 9.

THE CONTROL CONNECTION

An NFILE session is begun when the NFILE user program connects to a remote host and establishes a network connection. This initial connection is the control conection of the dialogue. If TCP is used as the underlying protocol, contact NFILE's well-known port, 59. If Chaos is used, use the contact name "NFILE".

The control connection is the vehicle used by the user to send its commands, and the server to send its command responses. These types of communication occur over the NFILE control connection:

- The user side sends NFILE commands.
- The server sends command responses.
- The server sends "notifications" and "asynchronous errors". See the section "NFILE Errors and Notifications", section 10.
- During resynchronization (a special circumstance) either the user or server sends a mark.

All commands, command responses, and other data flowing over the NFILE control connection are transmitted in the format of "top-level token lists". The control connection expects never to receive "loose tokens"; that is, tokens not contained in token lists.

DATA CONNECTIONS

Data connections are established and discarded at user request, by means of two NFILE commands: DATA-CONNECTION and UNDATA-CONNECTION. Each data connection is associated with a specific control connection, which is the same control connection that caused the data connection to be established.

Each data connection is composed of two "data channels". Each data channel is capable of sending data in one direction. The term "input channel" refers to the data channel that transmits data from the server to the user side; "output channel" refers to the data channel that transmits data from the user to the server side. Throughout the NFILE documentation, the terms "input channel" and "output channel" are seen from the perspective of the user side. A single data channel can be used for one data transfer after another.

The format of the data transferred on the data channels is defined as a "token list data stream". See the section "Token List Data Stream", section 11.3. When the end of data is reached, the keyword token EOF is sent. Occasionally, token lists are transmitted over the data channels, such as asynchronous error descriptions.

5. NFILE FILE OPENING MODES

The NFILE OPEN command opens a file for reading, writing, or "direct access" at the server host. That means, in general, asking the host file system to access the file and obtaining a file number, pointer, or other quantity for subsequent rapid access to the file; this is called an "opening". The term "opening" translates to a file stream in Symbolics terminology, a JFN in TOPS-20 terminology, and a file descriptor in UNIX terminology. An opening usually keeps track of how many bytes have been read or written, and other bookkeeping information.

NFILE supports two ways of transferring file data, "data stream mode" and "direct access mode". A single mode is associated with each opening. Note that an NFILE dialogue can have more than one opening, and thus use both modes.

DATA STREAM MODE:

Data stream mode of file transfer is the default mode of NFILE's OPEN command. Data stream mode is appropriate when the entire file is transferred, either from user to server, or from server to user. Data stream mode is used more often than direct access mode.

The OPEN command includes a "handle" argument, which identifies the data channel to be used to transfer the data. The handle is used in subsequent commands to reference this particular opening. When a data stream opening is requested with the OPEN command, the file is opened and the data begins to flow immediately.

The sending side transmits the entire contents of the specified file over the specified data channel as rapidly as the network permits. When the sending side reaches the end of the file, it transmits a special control token to signal end of file. The receiving side expects an uninterrupted stream of bytes to appear immediately on its side of the data channel.

The user gives the CLOSE command to terminate a data stream transfer. CLOSE results in closing the file.

DIRECT ACCESS MODE:

Direct access mode enables reading or writing data from a given starting point in a file through a specified number of bytes. In direct access mode, data is requested and sent in individual transactions. To request a direct access mode opening, the OPEN command is used with a DIRECT-FILE-ID argument. (In data stream mode, no DIRECT-FILE-ID is supplied.) The direct file identifier is used in subsequent commands to reference the direct access opening.

When a file is opened in direct access mode, the flow of data does not start immediately. Rather, the user gives either a READ command (to request data to flow from server to user) or a DIRECT-OUTPUT command (to request data to flow from user to server). When reading, the READ command allows the user to specify the starting point and the number of bytes of data to transfer. When writing, the FILEPOS command can be used to specify the starting point, before the DIRECT-OUTPUT command is given. The user can give many READ and DIRECT-OUTPUT commands, one after another.

The user side terminates the direct access transfer by using the CLOSE command. The ABORT command can be given to terminate without transmitting all of the specified bytes.

6. NFILE CHARACTER SET

The NFILE character set <1> is an extension of standard ASCII. NFILE command and response names use only the standard ASCII characters. However, the protocol supports the transfer of the non-ASCII characters in the NFILE character set; these characters might be stored in files, or might be used in pathnames.

Servers on machines that do not natively use the NFILE character set must perform character set translations for character openings, depending on the requested translation mode. No translation is required for binary openings. There are three translation modes for character openings: NORMAL, RAW, and SUPER-IMAGE. Each mode specifies a way to translate between the server's native set and the NFILE character set.

NORMAL mode is the default of the OPEN command. The translation for NORMAL mode ensures that a file containing characters in the NFILE character set can be written to a remote host and read back intact. OPEN has optional keyword arguments to specify RAW or SUPER-IMAGE. RAW mode means to perform no translation whatsoever. SUPER-IMAGE mode is intended for use by PDP-10 family machines only. It is included largely as an illustration of a system-dependent extension.

The details of each translation mode are given in Appendices:

> See the section "NORMAL Translation Mode", Appendix A. See the section "RAW Translation Mode", Appendix B. See the section "SUPER-IMAGE Translation Mode", Appendix C.

The use of the NFILE character set brings up a difficulty involved with determining an exact position within a character file. Some NFILE characters expand to more than one native character on some servers. Thus, for character files, when we speak of a given position in a file or the length of a file, we must specify whether we are speaking in "NFILE units" or "server units", because the counting of characters is different. This causes major problems in file position reckoning for character files. Specifically, it is futile for a user side to carefully monitor file position during output by counting characters, when character translation is in effect. The server's operating system interface for "position to point x in a file" necessarily operates in server units, but the user side has counted in NFILE units. The user side cannot try to second-guess the translation-counting process without losing host-independence. See the section "FILEPOS NFILE Command".

7. CONVENTIONS USED IN THIS DOCUMENT

7.1 Mapping Data Types Into Token List Representation

Throughout this NFILE specification, the data types of arguments, return values, asynchronous error descriptions, and notifications are described as being strings, integers, dates, time intervals, and so on. However, each conceptual data type must be mapped into the appropriate token list representation for transmission. The mapping of conceptual data types to token list representation is shown here:

Conceptual Type	Token List Representation
Keyword	A keyword token
Keyword list	A token list of keyword tokens
Integer	A numeric data token
String	A data token containing the characters of the string in the NFILE character set.
Boolean Truth	The token known as BOOLEAN-TRUTH.
Boolean False	The empty token list.
Date	A numeric data token. The date is expressed in Universal Time format, which measures a time as the number of seconds since January 1, 1900, at midnight GMT.
Date-or-never	Can be either a date or the empty token list, representing "never". "Never" is used for values such as the time a directory was last expunged, if it has never been expunged.
Time interval	A numeric data token. The time interval is expressed in seconds. A time interval indicating "never" is represented by the empty token list.

7.2 Format of NFILE Commands and Responses

Each command description begins by giving the command format and response format. Here is the beginning of the DATA-CONNECTION command description:

Command: (DATA-CONNECTION tid new-input-handle new-output-handle)

Response: (DATA-CONNECTION tid connection-identifier)

The command descriptions follow these conventions:

1. NFILE commands and responses are transmitted as top-level token lists.

   Top-level token lists are enclosed in parentheses in these command descriptions. These parentheses are not sent literally across the control or data connections, but are a shorthand representation of special control tokens that delimit top-level token lists. Specifically, TOP-LEVEL-LIST-BEGIN starts a top-level token list; TOP-LEVEL-LIST-END ends a top-level token list.

2. NFILE command names are keywords.

   The command name is required in every command and command response. All NFILE command names are keywords. Keywords appear in the NFILE documentation as their names in uppercase. For example, DATA-CONNECTION and DELETE are two command names.

3. A unique transaction identifier (tid) identifies each command.

   The transaction identifier is a string made up by the user side to identify this particular transaction, which is composed of the command and the response associated with this command. The transaction identifier is abbreviated in the command descriptions as tid. Transaction identifiers are limited to fifteen characters in length. The transaction identifier is required in every command and command response.

OPTIONAL ARGUMENTS

Many NFILE commands have "optional arguments". Optional arguments can be supplied (with appropriate values), or left out. If optional arguments are left out, their omission must be made explicit by means of substituting the empty token list in their place. The only exception to that rule is for trailing optional arguments or return values, which can be omitted without including the empty token list.

For example, the text of the DELETE command description explains that either a handle or a pathname must be supplied, but not both; therefore, one of them is an optional argument. Here is the command format of DELETE:

    (DELETE tid handle pathname)

If you supply a handle and no pathname, the command format is:

(DELETE tid handle)

If you supply a pathname and no handle, the command format is:

(DELETE tid empty-token-list pathname)

The empty token list in the token list stream appears as a LIST-BEGIN followed immediately by a LIST-END.

OPTIONAL KEYWORD/VALUE PAIRS

Four NFILE commands have "optional keyword/value pairs". These commands are: COMPLETE, LOGIN, OPEN, and READ. Optional keyword/value pairs can be either included in the command or omitted entirely. There is no need to substitute the empty token list for ommitted optional keyword tokens, unlike optional arguments. The order of the option keyword/value pairs is not significant.

If included, optional keyword/value pairs are a sequence of alternating keywords and values. The values associated with the keywords can be keywords, lists, strings, Booleans, integers, dates, date-or-never's, and time intervals. The text of each command description states what type of value is appropriate for each optional keyword.

Optional keyword/value pairs appear in the text as the keyword only, in uppercase letters. For example, here is the format of the LOGIN command:

Command Format:

(LOGIN tid user password FILE-SYSTEM USER-VERSION)

FILE-SYSTEM and USER-VERSION are two optional keywords associated with the LOGIN command. The user side can supply USER-VERSION, and omit FILE-SYSTEM as shown in this example:

(LOGIN x105 tjones let-me-in USER-VERSION 2)

As seen above, the optional keyword/value pair USER-VERSION, if supplied in a command, consists of the keyword USER-VERSION followed by the value to be used for that keyword (in this example, 2).

7.3 Data Channel Handles and Direct File Identifiers

Several NFILE commands require an argument that specifies an opening. This kind of argument is called a handle in the command description. It is always a string type argument. A handle can be either a data channel handle or a direct file identifier, depending on the mode of the opening:

Data Stream

The handle must identify a data channel that is bound to an opening.

Direct Access

In general, the handle must be a direct file identifier. A direct file identifier specifies a direct access opening. It is the same as the value supplied in the DIRECT-FILE-ID keyword/value pair in the OPEN command. It is used for all operations that identify an opening rather than a data channel.

Two NFILE commands applicable to direct access openings are exceptions to the general rule. The handle supplied in ABORT and CONTINUE cannot be a direct file identifier, but must be a data channel handle instead.

7.4 Syntax of File and Directory Pathname Arguments

Some arguments and return values in the NFILE command descriptions represent file pathnames. These are strings in the pathname syntax native to the server host. These pathnames contain no host identifiers of any kind. These pathnames must be fully defaulted, in the sense that they have a directory and file name (and file type, if the server operating system supports file types). If appropriate, a device is referenced in the pathname. If the server file system supports version numbers, there is always an explicit version number, even if that number or other specification is that system's representation of "newest" or "oldest".

Here are some examples of file pathnames, for different server hosts:

Server Host	Example of File Pathname
UNIX	/usr/max/life.c
TOPS-20	ps:<max>life.bin.17
VMS	MACD:[MAX]LIFE.FOR;3
Symbolics LMFS	>max>life.lisp.newest

The CREATE-DIRECTORY and HOME-DIRECTORY commands take a directory as an argument. In NFILE commands, a directory is represented by a string that names the directory. In most cases this string is in the syntax native to the server host. However in some cases the native format is modified somewhat to clarify that the string names a directory, and not a file. For example, a directory on UNIX is represented by "/usr/max/", not "/usr/max".

Here are some examples of directory pathnames for different server hosts:

Server Host	Example of Directory Pathname
UNIX	/usr/max/
TOPS-20	<max>
VMS	MACD:[MAX]
Symbolics LMFS	>max>hacks>

7.5 Format of NFILE File Property/Value Pairs

Several NFILE commands request information regarding the properties of files or directories. These commands include: DIRECTORY, MULTIPLE-FILE-PLISTS, PROPERTIES, and CHANGE-PROPERTIES. This section describes how file property information is conveyed over the token list stream.

File property information is usually sent in property/value pairs, where the property identifies the property, and the following value gives the value of that property for the specified file.

Each property is denoted either by a keyword or an integer. You can mix both ways of specifying properties (keyword or integer) within a single description. An integer is interpreted as an index into the Property Index Table, an array of property keywords. The server can optionally send a Property Index Table to the user during the execution of the LOGIN command, although it is not required. This greatly reduces the length of transmissions.

In command arguments, file properties cannot be specified with integers; keywords must be used to specify file properties in command arguments. Integers can be used to denote file properties only in command responses.

We now list the keywords associated with file properties. This list is not intended to be restrictive. If a programmer implementing NFILE needs a new keyword, a new keyword (not on this list) can be invented. The type of value of any new keywords is by default string. The keywords are sorted here by conceptual data type:

Data type	Keywords denoting file properties
Integers	BLOCK-SIZE, BYTE-SIZE, GENERATION-RETENTION-COUNT, LENGTH-IN-BLOCKS, LENGTH-IN-BYTES, DEFAULT-GENERATION-RETENTION-COUNT
Dates	CREATION-DATE, MODIFICATION-DATE
Date-or-never's	REFERENCE-DATE, INCREMENTAL-DUMP-DATE, COMPLETE-DUMP-DATE, DATE-LAST-EXPUNGED, EXPIRATION-DATE
Time intervals	AUTO-EXPUNGE-INTERVAL
Keyword Lists	SETTABLE-PROPERTIES, LINK-TRANSPARENCIES, DEFAULT-LINK-TRANSPARENCIES
Boolean values	DELETED, DONT-DELETE, DONT-DUMP, DONT-REAP, SUPERSEDE-PROTECT, NOT-BACKED-UP, OFFLINE, TEMPORARY, CHARACTERS, DIRECTORY

Strings	ACCOUNT, AUTHOR, LINK-TO, PHYSICAL-VOLUME, PROTECTION, VOLUME-NAME, PACK-NUMBER, READER, DISK-SPACE-DESCRIPTION, and any keywords not on this list

Note that these keyword names are intended to imply the semantics of the properties. For a discussion of the semantics of CREATION-DATE: See the section "NFILE OPEN Response Return Values", section 8.20.2. The "Reference Guide to Streams, Files, and I/O" in the Symbolics documentation set details the semantics that Symbolics associates with these properties.

8. NFILE COMMANDS

It is important to understand the conventions used in each of the following command descriptions. See the section "Conventions Used in This Document", section 7.

8.1 ABORT Command

Command: (ABORT tid input-handle)

Response: (ABORT tid)

ABORT cleanly interrupts and prematurely terminates a single direct access mode data transfer initiated with READ. The required input-handle string argument identifies a data channel on which an input transfer is currently taking place; this must be a direct access transfer. input-handle must identify a data channel; it cannot be a direct file identifier.

Upon receiving the ABORT command, the server checks to see if a transfer is still active on that channel. If so, the server terminates the transfer by telling the data connection logical process to stop transferring bytes of data. The user side needs to issue this command only when there are outstanding unread bytes. This excludes the case of the data channel having been disestablished or reallocated by the user side.

Whether or not a transfer is active on that channel, the user side puts the data channel into the unsafe state. Before the data channel can be used again, it must be resynchronized.

8.2 CHANGE-PROPERTIES Command

Command: (CHANGE-PROPERTIES tid handle pathname property-pairs)

Response: (CHANGE-PROPERTIES tid)

CHANGE-PROPERTIES changes one or more properties of a file. Either a handle or a pathname must be given, but not both. Whichever one is given must be supplied as a string. handle identifies a data channel that is bound to an open file; it can be a direct file identifier. pathname identifies a file on the server machine.

property-pairs is a required token list of keyword/value pairs, where the name of the property to be changed is the keyword, and the desired new property value is the value.

The properties that can be changed are host-dependent, as are any restrictions on the values of those properties. The properties that can be changed are the same as those returned as settable-properties, in the command response for the PROPERTIES command.

The server tries to modify all the properties listed in property-pairs to the desired new values. There is currently no definition about what should be done if the server can successfully change some properties but not others.

For further information on file property keywords and associated values: See the section "Format of NFILE File Property/Value Pairs", section 7.5.

8.3 CLOSE Command

Command: (CLOSE tid handle abort-p)

Response: (CLOSE tid truename binary-p other-properties)

CLOSE terminates a data transfer, and frees a data channel. The handle must be a data channel handle for a data stream opening, or a direct file identifier for a direct access opening. If a data channel is given, a transfer must be active on that handle. If abort-p is supplied as Boolean truth, the file is close-aborted, as described below.

"Closing the file" has different implications specific to each operating system. It generally implies invalidation of the pointer or logical identifier obtained from the operating system when the file was "opened", and freeing of operating system and/or job resources associated with active file access. For output files, it involves ensuring that every last bit sent by the user has been successfully written to disk. The server should not send a successful response until all these things have completed successfully.

In either data stream or direct access mode, the user can request the server to close-abort the file, instead of simply closing it. To close-abort a file means to close it in such a way, if possible, that it is as if the file had never been opened. In the specific case of a file being created, it must appear as if the file had never been created. This might be more difficult to implement on certain operating systems than others, but tricks with temporary names and close-time renamings by the server can usually be used to implement close-abort in these cases. In the case of a file being appended to, close-abort means to forget the appended data.

AN UNSUCCESSFUL CLOSE OPERATION

For the normal CLOSE operation (not a close-abort), after writing every last bit sent by the user to disk, and before closing the file, the server checks the data channel specified by handle to see if an asynchronous error is outstanding on that channel. That is, the server must determine whether it has sent an asynchronous error description to the user, to which the user has not yet responded with a CONTINUE command. If so, the server is unable to close the file, and therefore sends a command error response indicating that an error is pending on the channel. The appropriate three-letter error code is EPC. See the section "NFILE Errors and Notifications", section 10.

A SUCCESSFUL CLOSE OPERATION

The return values for OPEN and CLOSE are syntactically identical, but the values might change between the time of the file being opened and when it is closed. For example, the truename return value is supplied after all the close-time renaming of output files is done and the version numbers resolved (for operating systems supporting version numbers). Therefore, on some systems the truename of a file has one value at the time it is opened, and a different value when it has been closed. For a description of the CLOSE return values: See the section "NFILE OPEN Response Return Values", section 8.20.2.

If the user gives the CLOSE command with abort-p supplied as Boolean truth, thus requesting a close-abort of the file, the server need not check whether an asynchronous error description is outstanding on the channel. The server simply close-aborts the file.

8.4 COMPLETE Command

Command: (COMPLETE tid string pathname DIRECTION NEW-OK DELETED)

Response: (COMPLETE tid new-string success)

COMPLETE performs file pathname completion.

string is a partial filename typed by the user and pathname is the default name against which it is being typed. Both string and pathname are required arguments, and are of type string. The remaining arguments are optional keyword/value pairs.

NEW-OK is Boolean; if followed by Boolean truth, the server should allow either a file that already exists, or a file that does not yet exist. The default of NEW-OK is false; that is, the server does not consider files that do not already exist.

DELETED is a Boolean type argument; if followed by Boolean truth, the server is instructed to look for files that have been deleted but not yet expunged, as well as non-deleted files. The default is to ignore soft-deleted files.

DIRECTION can be followed by READ, to indicate that the file is to be read. If the file is to be written, DIRECTION can be followed by WRITE. The default is READ.

The filename is completed according to the files present in the host file system, and the expanded string new-string is returned. New-string is always a string containing a file name: either the original string, or a new, more specific string. The value of success indicates the status of the completion. The keyword value OLD or NEW means complete success, whereas the empty token list means failure. The following values of success are possible:

Value	Meaning
OLD	Success: the string completed to the name of a file that exists.
NEW	Success: the string completed to the name of a file that could be created.
Empty token list	Failure due to one of these reasons: The file is on a file system that does not

support completion. new-string is supplied as the unchanged string.

There is no possible completion. new-string is supplied as the unchanged string.

There is more than one possible completion. The given string is completed up to the first point of ambiguity, and the result is supplied as new-string.

A directory name was completed. Completion was not successful because additional components to the right of this directory remain to be specified. The string is completed through the directory name and the delimiter that follows it, and the result is returned in new-string.

The semantics of COMPLETE are not documented here. See the "Reference Guide to Streams, Files, and I/O" in the Symbolics documentation set for the recommended semantics of COMPLETE.

8.5 CONTINUE Command

Command: (CONTINUE tid handle)

Response: (CONTINUE tid)

CONTINUE resumes a data transfer that was temporarily suspended due to an asynchronous error. Each asynchronous error description has an optional argument of RESTARTABLE, indicating whether it makes any sense to try to continue after this particular error occurred. CONTINUE tries to resume the data transfer if the error is potentially recoverable, according to the RESTARTABLE argument in the asynchronous error description. For a discussion of asynchronous errors: See the section "NFILE Errors and Notifications", section 10.

handle is a required string-type argument that refers to the handle of the data channel that received an asynchronous error. That data channel could have been in use for a data stream or direct access transfer. handle cannot be a direct file identifier.

If the asynchronous error description does not contain the RESTARTABLE argument, and the user issues the CONTINUE command anyway, the server gives a command error response.

8.6 CREATE-DIRECTORY Command

Command: (CREATE-DIRECTORY tid pathname property-pairs)

Response: (CREATE-DIRECTORY tid dir-truename)

CREATE-DIRECTORY creates a directory on the remote file system. The required pathname argument is a string identifying the pathname of the directory to be created. The return value dir-truename is the pathname of the directory that was successfully created. Both of these pathnames are directory pathnames: See the section "Syntax of File and Directory Pathname Arguments", section 7.4.

property-pairs is a keyword/value list of properties that further define the attributes of the directory to be created. The allowable keywords and associated values are operating system dependent; typically they indicate arguments to be given to the native primitive for creating directories.

If property-pairs is supplied as the empty token list, default access and creation attributes apply and should be assured by the server. See the section "Format of NFILE File Property/Value Pairs", section 7.5.

8.7 CREATE-LINK Command

Command: (CREATE-LINK tid pathname target-pathname properties)

Response: (CREATE-LINK tid link-truename)

CREATE-LINK creates a link on the remote file system.

pathname is the pathname of the link to be created; target-pathname is the place in the file system to which the link points. Both are required arguments. The return value link-truename names the resulting link.

If a server on a file system that does not support links receives the CREATE-LINK command, it sends a command error response.

The arguments pathname and target-pathname, and the return value link-truename, are all strings in the full pathname syntax of the server host. See the section "Syntax of File and Directory Pathname Arguments", section 7.4.

The required properties argument is a token list of keyword/value pairs. These properties and their values specify certain attributes to be given to the link. The allowable keywords and associated

values are operating system dependent; typically they indicate arguments to be given to the native primitive for creating links.

If no property pairs are given in the command, the server should apply a reasonable default set of attributes to the link. See the section "Format of NFILE File Property/Value Pairs", section 7.5.

8.8 DATA-CONNECTION Command

Command: (DATA-CONNECTION tid new-input-handle new-output-handle)

Response: (DATA-CONNECTION tid connection-identifier)

DATA-CONNECTION enablesthe user side to initiate the establishment of a new data connection. The user side supplies two required string arguments, new-input-handle and new-output-handle. These arguments are used by subsequent commands to reference the two data channels that constitute the data connection now being created. new-input-handle describes the server-to-user data channel, and new-output-handle describes the user-to-server channel. new-input-handle and new-output-handle cannot refer to any data channels already in use.

Upon receiving the DATA-CONNECTION command, the server arranges for a logical port (called socket or contact name on some networks) to be made available on the foreign host machine. When the server has made that port available, it must inform the user of its identity. The server relays that information in the command response, in the required connection-identifier, a string. The server then listens on the port named by connection-identifier, and waits for the user side to connect to it.

Upon receiving the success command response, the user side supplies the connection-identifier to the local network implementation, in order to connect to the specified port. The data connection is not fully established until the user side connects successfully to that port. This command is unusual in that the successful command response does not signify the completion of the command; it indicates only that the server has fulfilled its responsibility in the process of establishing a data connection.

The connection-identifier informs the user of the correct identity of the logical port that the server has provided. NFILE expects the connection-identifier to be a string. For TCP this string is the port number represented in decimal. For Chaosnet, this string is the contact name. The connection-identifier is used only once; in all subsequent NFILE commands that need to reference either of the data channels that constitute this data connection, the new-input-handle and new-output-handle are used.

For background information: See the section "NFILE Control and Data Connections", section 4.

8.9 DELETE Command

Command: (DELETE tid handle pathname)

Response: (DELETE tid)

DELETE deletes a file on the remote file system.

Either a handle or a pathname must be supplied, but not both. If given, the handle must be a data channel handle for a data stream opening, or a direct file identifier for a direct access opening. pathname is a string in the full pathname syntax of the server host. See the section "Syntax of File and Directory Pathname Arguments", section 7.4.

With a pathname supplied, the DELETE command causes the specified file to be deleted. DELETE has different results depending on the operating system involved. That is, DELETE causes soft deletion on TOPS-20 and LMFS, and hard deletion on UNIX and Multics. If an attempt is made to delete a delete-through link on a Symbolics LMFS, its target is deleted instead.

If the handle argument is supplied to DELETE, the server deletes the open file bound to the data channel specified by handle at close time. This is true in both the output and input cases.

8.10 DIRECT-OUTPUT Command

Command: (DIRECT-OUTPUT tid direct-handle output-handle)

Response: (DIRECT-OUTPUT tid)

DIRECT-OUTPUT starts and stops output data flow for a direct access file opening. DIRECT-OUTPUT explicitly controls binding and unbinding of an output data channel to a direct access opening.

direct-handle is a required argument, and output-handle is optional.

If supplied, output-handle is a request to bind an output data channel (indicated by output-handle) to the direct access opening designated by the direct-handle. The specified output data channel must be free. The server binds the data channel and begins accepting data from that connection and writing it to the opening.

If the output-handle is omitted, this is a request to unbind the channel and terminate the active output transfer.

8.11 DIRECTORY Command

Command: (DIRECTORY tid input-handle pathname control-keywords properties)

Response: (DIRECTORY tid)

DIRECTORY returns a directory listing including the identities and attributes for logically related groups of files, directories, and links. If the command is successful, a single token list containing the requested information is sent over the data channel specified by input-handle, and the data channel is then implicitly freed by both sides <2>. For details on the format of the token list: See the section "NFILE DIRECTORY Data Format", section 8.11.1.

pathname specifies the files that are to be described; it is a string in the full pathname syntax of the server host. See the section "Syntax of File and Directory Pathname Arguments", section 7.4.

The pathname generally contains wildcard characters, in operating-system-specific format, describing potential file name matches. Most operating systems provide a facility that accepts such a pathname and returns information about all files matching this pathname. Some operating systems allow wildcard (potential multiple) matches in the directory or device portions of the pathname; other operating systems do not. There is no clear contract at this time about what is expected of servers on systems that do not allow wildcard matches (or some kinds of wild card matches), when presented with a wildcard.

properties is a token list of keywords that are the names of properties. If properties is omitted or supplied as the empty token list, the server sends along all properties. If any properties are supplied, the user is requesting the server to send only those properties.

control-keywords ARGUMENT TO DIRECTORY

control-keywords is a token list of keywords. The control-keywords affect the way the DIRECTORY command works on the server machine. Although some of the options below request the server to limit (by some filter) the data to be returned, it is never an error if the server returns more information than is requested.

The following keywords are recognized:

DELETED

Includes soft-deleted files in the directory list. Without this option, they must not be included. Such files have the DELETED property indicated as true" among their properties. DELETED is ignored on systems that do not support soft deletion.

DIRECTORIES-ONLY

This option changes the semantics of DIRECTORY fairly drastically. Normally, the server returns information about all files, directories, and links whose pathnames match the supplied pathname. This means that for each file, directory, or link to be listed, its directory name must match the potentially wildcarded) directory name in the supplied pathname, its file name must match the file name in the supplied pathname, and so on.

When DIRECTORIES-ONLY is supplied, the server is to list only directories, not whose pathnames match the supplied pathname, but whose pathnames expressed as directory pathnames match the (potentially wildcarded) directory portion of the supplied pathname. The description of the PROBE-DIRECTORY keyword that can be supplied as the direction argument of the OPEN command discusses this: See the section "OPEN Command", section 8.20.

It is not yet established what servers on hosts that do not support this type of action natively are to do when presented with DIRECTORIES-ONLY and a pathname with a wildcard directory component.

FAST Speeds up the operation and data transmission by not listing any properties at all for the files concerned; that is, only the truenames are returned.

NO-EXTRA-INFO

Specifies that the server is to suppress listing those properties that are generally more difficult or expensive to obtain. This typically eliminates listing of directory-specific properties such as information about default generation counts and expunge dates.

SORTED

This causes the directory listing to be sorted. The sorting is done alphabetically by directory, then by file name, then file type, then file version (by increasing version number).

8.11.1 NFILE DIRECTORY Data Format

If the NFILE DIRECTORY command completes successfully, a single token list containing the requested directory information is sent on the data channel specified by the input-handle argument in the DIRECTORY command. This section describes the format of that single token list, and gives further detail on the properties argument to DIRECTORY.

The token list is a top-level token list, so it is delimited by TOP-LEVEL-LIST-BEGIN and TOP-LEVEL-LIST-END. The top-level token list contains embedded token lists. The first embedded token list contains the empty token list followed by property/value pairs describing property information of the file system as a whole rather than of a specific file. NFILE requires one property of the file system to be present: DISK-SPACE-DESCRIPTION is a string describing the amount of free file space available on the system. The following embedded token lists contain the pathname of a file, followed by property/value pairs describing the properties of that file.

The following example shows the format of the top-level token list returned by DIRECTORY, for two files. It is expected that the server return several property/value pairs for each file; the number of pairs returned is not constrained. In this example, two property/value pairs are returned for the file system, two pairs are returned for the first file, and only one pair is returned for the second file.

```
TOP-LEVEL-LIST-BEGIN
LIST-BEGIN          - first embedded token list starts
LIST-BEGIN          - an empty embedded token list starts
LIST-END            - the empty embedded token list ends
prop1 value1        - property/value pairs of file system
prop2 value2
LIST-END
```

```
LIST-BEGIN
pathname1          - pathname of the first file
prop1 value1       - property/value pairs of first file
prop2 value2
LIST-END
LIST-BEGIN
pathname2          - pathname of the second file
prop1 value1       - property/value pairs of second file
LIST-END
TOP-LEVEL-LIST-END
```

The following example is designed to illustrate the structure of the top-level token list by depicting TOP-LEVEL-LIST-BEGIN and TOP-LEVEL-LIST-END by parentheses and LIST-BEGIN and LIST-END by squarbe rackets. respectively. The indentation, blank spaces, and newlines in the example are not part of the token list, but are used here to make the structure of the token list clear.

```
([   [ ]    prop1 value1 prop2 value2]
 [pathname1 prop1 value1 prop2 value2]
 [pathname2 prop1 value1])
```

The pathname is a string in the full pathname syntax of the server host. See the section "Syntax of File and Directory Pathname Arguments", section 7.4.

For further information on file property/value pairs: See the section "Format of NFILE File Property/Value Pairs", section 7.5.

8.12 DISABLE-CAPABILITIES Command

Command: (DISABLE-CAPABILITIES tid capability)

Response: (DISABLE-CAPABILITIES tid cap-1 success-1
cap-2 success-2 cap-3 success-3 ...)

DISABLE-CAPABILITIES causes an access capability to be disabled on the server machine. capability is a string naming the capability to be disabled. The meaning of the capability is dependent on the operating system.

The return values cap-1, cap-2, and so on, are strings specifying names of capabilities. If the capability named by cap-1 was successfully disabled, the corresponding success-1 is supplied as Boolean truth; otherwise it is the empty token list.

Although the user can specify only one capability to disable, it is conceivable that the result of disabling that particular capability is the disabling of other, related capabilities. That is why the command response can contain information on more than one capability.

8.13 ENABLE-CAPABILITIES Command

Command: (ENABLE-CAPABILITIES tid capability password)}

Response: (ENABLE-CAPABILITIES tid cap-1 success-1
cap-2 success-2 cap-3 success-3 ...)

ENABLE-CAPABILITIES causes an access capability to be enabled on the server machine. The password argument is optional, and should be included only if it is needed to enable this particular capability. Both password and capability are strings. The meaning of the capability is dependent on the operating system.

The return values cap-1, cap-2 and so on, are strings specifying names of capabilities. If the capability named by cap-1 was successfully enabled, the corresponding success-1 is supplied as Boolean truth; otherwise it is the empty token list.

Although the user can specify only one capability to enable, it is conceivable that the result of enabling that particular capability is the enabling of other, related capabilities. That is why the command response can contain information on more than one capability.

8.14 EXPUNGE Command

Command: (EXPUNGE tid directory-pathname)

Response: (EXPUNGE tid server-storage-units-freed)

EXPUNGE causes the directory specified by pathname to be expunged. Expunging means that any files that have been soft deleted are to be permanently removed.

For file systems that do not support soft deletion, the command is to be ignored; a success command response is sent, but no action is performed on the file system. In this case, the number-of-server-storage-units-freed return value should be omitted.

directory-pathname is a required string argument in the directory pathname format; it must refer to a directory on the server file system, and not to a file. See the section "Syntax of File and Directory Pathname Arguments", section 7.4.

The return value server-storage-units-freed is an integer specifying how many records, blocks, or whatever unit is used to measure file storage on the server host system, were recovered. This return value should be omitted if the server does not know how many storage units were freed.

The protocol does not define whether directory-pathname is really a pathname as directory or a wildcard pathname of files to be expunged. The protocol does not define whether or not wildcards are permitted, or required to be supported, in the directory portion of the pathname (representing an implicit request to expunge many directories).

8.15 FILEPOS Command

Command: (FILEPOS tid handle position resync-uid)

Response: (FILEPOS tid)

FILEPOS sets the file access pointer to a given position, relative to the beginning of the file. FILEPOS is used to indicate the position of the next byte of data to be transferred.

The handle indicates the file to be affected. handle must be a data channel handle for a data stream opening, or a direct file identifier for a direct access opening. Both handle and position are required arguments.

position is an integer indicating to which point in the file the file access pointer is to be reset. position is either a byte number according to the current byte size being used, or characters for character openings. Position zero is the beginning of the file. If this is a character opening, position is measured in server units, not in NFILE character set units.

If the FILEPOS command is given on an input data channel (that is, a data channel currently sending data from server to user), the affected data channel must be resynchronized after the FILEPOS is accomplished, in order to identify the start of the new data. The resync-uid is a unique identifier associated with the resynchronization of the data channel; it is unique with respect to this dialogue. resync-uid must be supplied if handle is an input handle, but it is not supplied otherwise. For more information on the resynchronization procedure: See the section "NFILE Data Connection Resynchronization", section 9.2.

In the output case, the user must somehow indicate to the server, on the output data channel, when there is no more data. The user side sends the keyword token EOF to do so. Upon receiving that control

token, the server is required to position the file pointer according to the position given. When the new file position is established, the server resumes accepting data at the new file position.

In most cases, using the direct access mode of transfer is more convenient and efficient than repeated use of FILEPOS with a data stream opening.

There are problems inherent in trying to set a file position of a character-oriented file on a foreign host, if one machine is a Symbolics computer and the other is not. For example, character set translation must take place. See the section "NFILE Character Set", section 6. Because of these difficulties, FILEPOS might not be supported in the future on character files. FILEPOS is not problematic for binary files.

8.15.1 Implementation Hint for FILEPOS Command

The server processing of this command (by the control connection handler) must not attempt to wait for the resynchronization procedure to complete. It is possible that the user could abort between sending the FILEPOS command and reading for the mark and resynchronization identifier. That scenario could leave the sender of the resynchronization identifier, on the server side, blocked for output indefinitely.

Only two commands received on the control connection can break the data channel out of the blocked state described above: CLOSE with abort-p supplied as Boolean truth, and RESYNCHRONIZE-DATA-CHANNEL. Therefore, the control connection must not wait for the data channel to finish performing the resynchronization procedure. This wait should instead be performed by the process managing the data channel.

8.16 FINISH Command

Command: (FINISH tid handle)

Response: (FINISH tid truename binary-p other-properties)

FINISH closes a file and reopens it immediately with the file position pointer saved, thus leaving it open for further I/O. If possible, the implementation should do the closing and opening in an indivisible operation, such that no other process can get access to the file.

The arguments, results, and their meaning are identical to those of the CLOSE command. See the section "CLOSE Command", section 8.3. FINISH requires a handle, which has the same meaning as the handle of the CLOSE command.

In the output case, for both direct mode and data stream mode of openings, the server writes out all buffers and sets the byte count of the file. The user sends the keyword token EOF on the data channel, to indicate that the end of data has been reached. The server leaves the file in such a state that if the system or server crashes anytime after the FINISH command has completed, it would later appear as though the file had been closed by this command. However, the file is not left in a closed state now; it is left open for further I/O operations. FINISH is a reliability feature.

FINISH is somewhat pointless in the input case, but valid. The native Symbolics file system (LMFS) implements FINISH on an output file by an internal operation that effectively goes through the work of closing but leaves the file open for appending.

ERRORS ON FINISH

After writing every last bit sent by the user to disk, and before closing the file, the server checks the data channel specified by handle to see if an asynchronous error is outstanding on that channel. That is, the server must determine whether it has sent an asynchronous error to the user, to which the user has not yet responded with a CONTINUE command. If so, the server is unable to finish the file, and it must send a command error response response, indicating that an error is pending on the channel. The appropriate three-letter error code is EPC. See the section "NFILE Errors and Notifications", section 10.

8.17 HOME-DIRECTORY Command

Command: (HOME-DIRECTORY tid user)

Response: (HOME-DIRECTORY tid directory-pathname)

HOME-DIRECTORY returns the full pathname of the home directory on the server machine for the given user.

user is a string that should be recognizable as a user's login name on the server operating system. directory-pathname is a string in the directory pathname format. See the section "Syntax of File and Directory Pathname Arguments", section 7.4.

8.18 LOGIN Command

Command: (LOGIN tid user password FILE-SYSTEM USER-VERSION)

Response: (LOGIN tid keyword/value-pairs)

LOGIN logs the given user in to the server machine, using the password if necessary. Both user and password are string arguments; user is required, password is optional. An omitted password is valid if the host allows the specified user to log in without a password. Depending on the operating system and server, it might be necessary to log in to run a program (in this case the NFILE server program) on the host. LOGIN establishes a user identity that is used by the operating system to establish the file author and determine file access rights during the current session.

The server has the option to reject with an error any command except LOGIN if a successful LOGIN command has not been performed. This is recommended. Many operating systems perform the login function in a different process and/or environment than user programs. The portion of the NFILE server running in the special login environment could conceivably be capable only of processing the LOGIN command; this is the reason for having the LOGIN command in NFILE.

FILE-SYSTEM and USER-VERSION are optional keyword/value pairs. The FILE-SYSTEM keyword/value pair selects the identity of the file system to which all following commands in this session are to be directed. This argument has meaning only if the server host machine has multiple file systems, and the targeted file system is other than the default file system that a user would get by initiating a dialogue with that host. The FILE-SYSTEM argument is an arbitrary token list. If the server does not recognize it, the server gives an appropriate command error response.

Currently, the only use of FILE-SYSTEM is for Symbolics servers to select one of the front-end processor hosts instead of the LMFS, which is the default. In this case, the first element in the token list is the keyword FEP, and the second element in the token list is an integer, indicating the desired FEP disk unit number. If the server discovers there is no such file system, the server gives a command error response including the three-letter code NFS, meaning "no file system". See the section "NFILE Errors and Notifications", section 10.

The user tells the server what version of NFILE it is running by including the optional USER-VERSION keyword/value pair. The value associated with USER-VERSION can be a string, an integer, or a token list. This document describes NFILE user version 2 and server version 2.

Upon receiving the representation of the user version, the server can either adjust certain parameters to handle this particular version, or simply ignore the user version altogether. Currently, the only released versions of NFILE are user version 2 and server version 2.

LOGIN RETURN VALUES: keyword/value-pairs

The keyword/value-pairs is a token list composed of keywords followed by their values. The server includes any or all of the following keywords and their values; they are all optional. The following keywords are recognized:

NAME

The value associated with NAME is a string specifying the user identity, in the server host's terms.

PERSONAL-NAME

The value associated with PERSONAL-NAME is a string representing the user's personal name, last name first. For example: "McGillicuddy, Aloysius X.".

HOMEDIR-PATHNAME

The value associated with HOMEDIR-PATHNAME is a string in the pathname as directory format, indicating the home directory of the user. See the section "Syntax of File and Directory Pathname Arguments", section 7.4.

GROUP-AFFILIATION

The value associated with GROUP-AFFILIATION is a string specifying the group to which the user belongs, when this concept is appropriate.

SERVER-VERSION

The value associated with SERVER-VERSION can be a string, an integer, or a token list. The value is a representation of the version of the server is running. Upon receiving the server version, the user can: adjust certain parameters to handle this particular version; accept

the version; or close the connection. Currently, the only released versions of NFILE are user version 2 and server version 2.

PROPERTY-INDEX-TABLE

The value associated with PROPERTY-INDEX-TABLE is a token list of keywords. This return value enables the server to inform the user which file properties are meaningful on its file system. The keywords in PROPERTY-INDEX-TABLE can be used by the DIRECTORY command (a user request for information on file properties of a specified directory or directories). The server can specify a certain property by giving an integer that is the index of that file property into the PROPERTY-INDEX-TABLE. This reduces the volume of data sent during directory listings. The first element in PROPERTY-INDEX-TABLE is indexed by the number 0. See the section "DIRECTORY Command", section 8.11.

8.19 MULTIPLE-FILE-PLISTS Command

Command: (MULTIPLE-FILE-PLISTS tid input-handle paths characters properties)

Response: (MULTIPLE-FILE-PLISTS tid)

MULTIPLE-FILE-PLISTS returns file property information of one or more files. The server sends the information in a data structure (the format is described later in this section) on the given input-handle. paths is an embedded token list composed of the pathnames in which the user is interested. Each pathname in this list is a string in the full pathname syntax of the server host. Unlike for the DIRECTORY command, wildcards are not allowed in these pathnames. See the section "Syntax of File and Directory Pathname Arguments", section 7.4.

characters is either Boolean truth (indicating that each file is a character file), the empty token list (each file is a binary file), or the keyword DEFAULT. DEFAULT indicates that the server itself is to figure out whether a file is a character or binary file. For more information on the meaning of the DEFAULT keyword: See the section "OPEN Command", section 8.20. The value of characters can influence some servers' idea of a file's length.

properties is a token list of keywords indicating which properties the user wants returned. The server is always free to return more properties than those requested in the properties argument. If properties is supplied as the empty token list, the server should transmit all known properties on the files.

The server transmits as much of the requested information as possible on the given input-handle. The information is contained in a top-level token list of elements. Each element corresponds with a supplied pathname; the order of the original pathlist must be retained in the returned token list. An element is an empty token list if the corresponding file or any of its containing directories does not exist. The elements that correspond to successfully located files are lists composed of truename followed by any properties. properties are keyword/value pairs. truename is a string in the full pathname syntax of the server host.

The following example shows TOP-LEVEL-LIST-BEGIN and TOP-LEVEL-LIST-END as parentheses, and LIST-BEGIN and LIST-END with square brackets.

For example, the user supplied a pathlist argument resembling:

[file1 file2 file3]

The server could not locate file1 or file3, but did locate file2, and found the length and author of file2. The top-level token list transmitted by the server is:

([] [truename-of-file2 LENGTH 381 AUTHOR williams] [])

For further detail on how file properties and values are expressed: See the section "Format of NFILE File Property/Value Pairs", section 7.5.

8.20 OPEN Command

Command: (OPEN tid handle pathname direction binary-p
TEMPORARY RAW SUPER-IMAGE DELETED PRESERVE-DATES
SUBMIT DIRECT-FILE-ID ESTIMATED-LENGTH BYTE-SIZE
IF-EXISTS IF-DOES-NOT-EXIST)

Response: (OPEN tid truename binary-p other-properties)

OPEN opens a file for reading, writing, or direct access at the server host. That means, in general, asking the host file system to access the file and obtaining a file number, pointer, or other quantity for subsequent rapid access to the file; this is called an "opening". See the section "NFILE File Opening Modes", section 5.

The OPEN command has the most complicated syntax of any NFILE command. The OPEN command has required arguments, an optional argument, and many optional keyword/value pairs. For details on the syntax of each of these parts of the OPEN command: See the section "Conventions Used in This Document", section 7.

The following arguments are required: pathname, direction, and binary-p. handle is an optional argument, which must either be supplied or explicitly omitted by means of substituting in its place the empty token list.

The OPEN command has many optional keyword/value pairs, which encode conceptual arguments to the server file system for the OPEN operation. A detailed description of all the supported OPEN optional keywords is given below.

The OPEN return values reflect information about the file opened, when the opening is successful. In the case of a probe-type opening, this information is returned when the given file (or link, or directory) exists and is accessible, even though the file (or link, or directory) is not actually opened. For detail on the OPEN return values: See the section "NFILE OPEN Response Return Values", section 8.20.2.

THE pathname OPEN ARGUMENT

The pathname is a required argument specifying the file to be opened. pathname is a string in the full pathname syntax of the server host. See the section "Syntax of File and Directory Pathname Arguments", section 7.4.

For some purposes (for example, when the OPEN argument direction is supplied as PROBE-DIRECTORY), only the directory specified by this pathname is utilized. See the section "NFILE OPEN Optional Keyword/Value Pairs", section 8.20.1.

THE handle OPEN ARGUMENT

The handle argument of the OPEN command specifies a data channel to be used for the transfer. Subsequent commands in this session use the same handle to specify this opening. It is the user side's responsibility to ensure that handle refers to an existing and free data channel that does not require resynchronization before use. A handle must be supplied, unless a probe-type opening is desired (that is, the direction is supplied as PROBE, PROBE-DIRECTORY, or PROBE-LINK) or a direct access opening is being requested (that is, a DIRECT-FILE-ID is supplied). In those cases, the empty token list is supplied for handle.

THE direction OPEN ARGUMENT

The direction argument must be supplied as one of these keywords: INPUT, OUTPUT, IO, PROBE, PROBE-DIRECTORY, and PROBE-LINK. The meanings of the direction keywords are as follows:

INPUT

Specifies that the file is to be opened for input server-to-user transfer). To request a direct access opening, supply a value for DIRECT-FILE-ID. If no DIRECT-FILE-ID is supplied, the opening is a data stream opening.

OUTPUT

Specifies that the file is to be opened for output user-to-server transfer). To request a direct access opening, supply a value for DIRECT-FILE-ID. If no DIRECT-FILE-ID is supplied, the opening is a data stream opening.

IO

Specifies that interspersed input and output will be performed on the file. This is only meaningful in direct access mode. A DIRECT-FILE-ID must also be supplied. See the section "NFILE OPEN Optional Keyword/Value Pairs", section 8.20.1.

If direction is supplied as PROBE, PROBE-LINK, or PROBE-DIRECTORY, the opening is said to be a probe-type opening. The DIRECT-FILE-ID option is meaningless and an error for probe-type openings. The file handle must be supplied as an empty token list for probe-type openings.

PROBE

Specifies that the file is not to be opened at all, but simply checked for existence. If the file does not exist or is not accessible, the error indications and actions are identical to those that would be given for an INPUT opening. If the file does exist, the successful command response contains the same information as it would have if the file had been opened for INPUT. If it is a link, the link is followed to its target.

PROBE-LINK

Like PROBE, with one difference. PROBE-LINK specifies that if the pathname is found to refer to a link, that link is not to be followed, and information about the link itself is to be returned.

PROBE-DIRECTORY

PROBE-DIRECTORY requests information about the directory designated by the pathname argument. In the PROBE-DIRECTORY case, the pathname argument refers to the directory on which information is requested. In all other cases, the pathname refers to a file to be opened. If pathname contains a file name and file type, these parts of the pathname are ignored for PROBE-DIRECTORY openings as long as they are syntactically valid.

THE binary-p OPEN ARGUMENT

The value of binary-p affects the mode in which the server opens the file, as well as informing it whether or not character set translation must be performed.

If binary-p is supplied as the empty token list, the opening is said to be a character opening. The server performs character set translation between its native character set and the NFILE character set. The data is transferred over the data connection one character per eight-bit byte. See the section "NFILE Character Set", section 6.

If binary-p is supplied as Boolean truth, the opening is said to be a binary opening. The user side supplies the byte size via the BYTE-SIZE option; if not supplied, the default byte size is 16 bits. If byte size is less than 9, the file data is transferred byte by byte. If the byte size is 9 or greater, the server transfers each byte of the file as two eight-bit bytes, low-order first.

binary-p can also be supplied as the keyword DEFAULT. DEFAULT specifies that the server itself is to determine whether to transfer binary or character data. DEFAULT is meaningful only for input openings; it is an error for OUTPUT, IO, or probe-type openings. For file systems that maintain the innate binary or character nature of a file, the server simply asks the file system which case is in force for the file specified by pathname.

When binary-p is supplied as DEFAULT, on file systems that do not maintain thisinformation, the server is required to perform a heuristic check for Symbolicsobject fileson the first two 16-bit bytes of the file. If the file isdetermined to be aSymbolics object file, the server performs a BINARY openingwith BYTE-SIZE of16; otherwise, it performs a CHARACTER opening.

The details of the check are as follows: if the first 16-bit byte is the octal number170023 and the second 16-bit byte is any number between 0 and 77 octal(inclusive), the file is recognized as a Symbolics object file. In any othercase, it is not.

8.20.1 NFILE OPEN Optional Keyword/Value Pairs

The OPEN command has many optional keyword/value pairs that encode conceptual arguments to the file system for the OPEN operation.

The following options are used often:

BYTE-SIZE

Must be followed by an integer between 1 and 16, inclusive, or the empty token list. BYTE-SIZE is meaningful only for binary openings. BYTE-SIZE can be ignored for probe-type openings. It can be omitted entirely for character openings, but if supplied, must be followed by the empty token list. If binary-p is supplied as DEFAULT, BYTE-SIZE can be omitted entirely, or followed by the empty token list.

If a binary opening is requested and BYTE-SIZE is not supplied, the assumed value is 16 for output openings. For input binary openings, the default is the host file system's stored conception of the file's byte size (for those hosts that natively support byte size). For file systems that do not natively support natively byte size, the default byte-size on binary input is 16.

For file systems that maintain the innate byte-size of each file, the server should supply this number to the appropriate operating system interface that performs the semantics of opening the file. For other operating systems, a file written with a given byte size must produce the same bytes in the same order when read with that byte size. In this case, the server or host operating system can choose any packing scheme that complies with this rule.

Operating systems that do not support byte size must ensure that binary files written from user ends of the current protocol can be read back correctly. However, the server can choose packing schemes that allow all bits of the server host's word to be accessed and concur with other packing schemes used by native host software.

For example, Multics supports 36 bit words and 9 bit bytes. A packing scheme appropriate for a Multics NFILE server is:

Byte Size	Packing Scheme
7, 8, or 9 bits	four per 36-bit word
10, 11, or 12 bits	three per 36-bit word
13, 14, 15, or 16 bits	two per 36-bit word

In the first packing scheme in the table, native Multics character-oriented software can access each logical byte sequentially. In the last packing scheme, each Symbolics byte is in a halfword, easily accessible and visible in an octal representation. To achieve maximum data transfer rate and access all bits of a Multics word, a byte size of 12 must be specified.

DELETED

If supplied as Boolean truth, DELETED specifies that deleted" files are to be treated as though they were not "deleted". DELETED is meaningful only for operating systems that support "soft deletion" and subsequent "undeletion" of files. Other operating systems must ignore this option. Normally, deleted files are not visible to the OPEN operation; this option makes them visible.

DELETED can also be followed by the empty token list, which has the same effect as omitting the DELETED keyword/value pair entirely. For output openings, DELETED is meaningless and an error if supplied.

DIRECT-FILE-ID

If supplied, the DIRECT-FILE-ID indicates that the opening is to be a direct access mode opening. If not supplied, the opening is a data stream opening. The value of DIRECT-FILE-ID is a string generated by the user, that has not been used as a DIRECT-FILE-ID in this dialogue, and does not designate any data channel. The DIRECT-FILE-ID is a unique identifier for the direct access opening. It is used for all operations that identify an opening rather than a data channel. The DIRECT-FILE-ID is used to identify a direct access opening, just as a file handle is used to identify a data stream opening. The PROPERTIES, CLOSE, and RENAME commands use the DIRECT-FILE-ID in this way. There are only two NFILE commands applicable to direct access openings (ABORT and CONTINUE) that do not use the DIRECT-FILE-ID, but use a data channel handle instead.

PRESERVE-DATES

If supplied as Boolean truth, PRESERVE-DATES specifies that the

server is to attempt to prevent the operating system from updating the "reference date" or date-time used" of the file. This is meaningful only for input openings, and is an error otherwise.

The Symbolics operating system invokes this option for operations such as View File in the editor, where it wishes to assert that the user did not "read" the file, but just "looked at it". Servers on operating systems that do not support reference dates or users revising or suppressing update of the reference dates must ignore this option.

ESTIMATED-LENGTH

The value of ESTIMATED-LENGTH is an integer estimating the length of the file to be transferred. This option is meaningful and permitted only for output openings. ESTIMATED-LENGTH enables the user end to suggest to the server's file system how long the file is going to be. This can be useful for file systems that must preallocate files or file maps or that accrue performance benefits from knowing this information at nthe time the file is first opened. This estimate, if supplied, is not required to be exact. It is ignored by servers to which it is not useful or interesting. The units of the estimate are characters for character openings, and bytes of the agreed-upon byte size for binary openings. The character units should be server units, if possible, but since this is only an estimate, NFILE character units are acceptable. See the section "NFILE Character Set", section 6.

IF-EXISTS

Meaningful only for output openings, ignored otherwise, but not diagnosed as an error. The value of IF-EXISTS is a keyword that specifies the action to Be taken if a file of the given name already exists. The semantics of the values are derived from the Common Lisp specification and repeated here for completeness. If the file does not already exist, the IF-EXISTS option and its value are ignored.

If the user side does not give the IF-EXISTS option, The action to be taken if a file of the given name already exists depends on whether or not the file system supports file versions. If it does, the default is ERROR (if an explicit version is given in the file pathname) or NEW-VERSION (if the version in the file pathname is the newest version). For file systems not supporting versions, the default is SUPERSEDE. These actions are described below.

IF-EXISTS provides the mechanism for overwriting or appending to files. With the default setting of IF-EXISTS, new files are created by every output opening.

Operating systems supporting soft deletion can take different actions if a "deleted" file already exists with the same name (and type and version, where appropriate) as a file to be created. The Symbolics file system (LMFS) effectively uses SUPERSEDE, even if not asked to do so. Other servers and file systems are urged to do similarly. Recommended action is to not allow deleted files to prevent successful file creation (with specific version number) even if an IF-EXISTS option weaker than SUPERSEDE, RENAME, or RENAME-AND-DELETE is specified or implied.

Here are the possible values and their meanings:

ERROR

Reports an error.

NEW-VERSION

Creates a new file with the same file name but with a larger version number. This is the default when the version component of the filename is newest. File systems without version numbers can implement this by effectively treating it as SUPERSEDE.

RENAME

Renames the existing file to some other name and then creates a new file with the specified name. On most file systems, this renaming happens at the time of a successful close.

RENAME-AND-DELETE

Renames the existing file to some other name and then deletes it (but does not expunge it, on those systems that distinguish deletion from expunging). Then it creates a new file with the specified name. On most file systems, this renaming happens at the time of a successful close.

OVERWRITE

Output operations on the opening destructively modify the existing file. New data replaces old data at the beginning of the file; however, the file is not truncated to length zero upon opening.

TRUNCATE

Output operations on the opening destructively modify the existing file. The file pointer is initially positioned at the beginning of the file; at that time, TRUNCATE truncates the file to length zero and frees disk storage occupied by it.

APPEND

Output operations on the opening destructively modify the existing file. New data is placed at the current end of the file.

SUPERSEDE

Supersedes the existing file. This means that the old file is removed or deleted and expunged. The new file takes its place. If possible, the file system does not destroy the old file until the new file is closed, against the possibility that the file will be close-aborted. This differs from NEW-VERSION in that SUPERSEDE creates a new file with the same name as the old one, rather than a file name with a higher version number.

There are currently no standards on what a server can do if it cannot implement some of these actions.

IF-DOES-NOT-EXIST

Meaningful for input openings, never meaningful for probe-type openings, and sometimes meaningful for output openings. IF-DOES-NOT-EXIST takes a value token, which specifies the action to be taken if the file does not already exist. Like IF-EXISTS, it is a derivative of Common Lisp. The default is as follows: If this is a probe-type opening or read opening, or if the IF-EXISTS option is specified as OVERWRITE, TRUNCATE, or APPEND, the default is ERROR. Otherwise, the default is CREATE.

These are the values for IF-DOES-NOT-EXIST:

ERROR

Reports an error.

CREATE

Creates an empty file with the specified name and then proceeds as if it already existed.

The following optional keyword/value pairs are rarely used, if ever:

RAW

If supplied as Boolean truth, RAW specifies that character set translation is not to be performed, but that characters are to be transferred intact, without inspection. This option is meaningful only for character openings; it is an error otherwise. It is also an error to supply RAW as Boolean truth for probe-type openings. RAW can also be followed by the empty token list, which has the same effect as if the RAW keyword/value pair were omitted entirely. See the section "RAW Translation Mode", Appendix B.

SUPER-IMAGE

If supplied as Boolean truth, SUPER-IMAGE specifies that Rubout quoting is not to be performed. This operation is meaningful only for character openings; it is an error otherwise. It is also an error for probe-type openings. SUPER-IMAGE can also be followed by the empty token list, which has the same effect as if the SUPER-IMAGE keyword/value pair were omitted entirely.

SUPER-IMAGE mode causes the server to read or write character files where ASCII Rubout characters are a significant part of the file content, not where they are an escape for this protocol. However, other translations must still be performed: See the section SUPER-IMAGE Translation Mode", Appendix C.

TEMPORARY

Used by the TOPS-20 server only. TEMPORARY says to use GJ%TMP in the GTJFN. This is useful mainly when writing files, and indicates that the foreign operating system is to treat the file as temporary. See TOPS-20 documentation for more about the implications of this option. Other servers can ignore it. This option is meaningless and an error for input or probe-type openings. TEMPORARY can also be followed by the empty token list, which has the same effect as if the TEMPORARY keyword/value pair were omitted entirely.

SUBMIT

SUBMIT is meaningful for output only. If supplied as Boolean truth, SUBMIT causes the server to submit the contents of the file being written to the operating system as a job, after the file is closed. VMS is an example of an operating system that could conveniently support SUBMIT. SUBMIT can also be followed by the empty token list, which has the same effect as if the SUBMIT

keyword/value pair were omitted entirely. Servers that do not implement this option should give an error response if requested to submit a file to the operating system.

8.20.2 NFILE OPEN Response Return Values

The results of a successful OPEN operation are reported in the command response. Here is the specification of the OPEN response format:

Response Format:

(OPEN tid truename binary-p other-properties)

The return values for OPEN and CLOSE are syntactically identical, but the values can change in the time interval between open and close.

truename is a string representing the pathname of the file in the full pathname syntax of the server host. It should be determined by the server once it has opened the file, via some request to its operating system. The request can be of the form: "What file corresponds to this JFN, file number, pointer, etc.?" If the operating system supports version numbers, this string always contains an explicit version number. It always contains a directory name, a file name, and so on.

Some operating systems might not know the truename of an output file until it is closed. It is permissible not to supply an explicit version number in the pathname in the OPEN response in this specific case. On these systems the truename when the file is opened is different than the truename after it has been closed.

The return value binary-p indicates whether the opening is a binary or character opening. For binary openings, binary-p is supplied as Boolean truth; for character openings it is the empty token list.

other-properties is a list of keyword/value pairs. other-properties must contain CREATION-DATE and LENGTH. AUTHOR should be included if the server operating system has a convenient mechanism for determining the author of the sfile. The other properties described here can be included if desired.

AUTHOR

The value of AUTHOR is a string representing the name of the author of the file. This is some kind of user identifier, whose format is system-specific. As with CREATION-DATE (see below), AUTHOR is supposed to represent the logical determinor of the current data

content of the file, not necessarily the agency that actually created the file.

BYTE-SIZE

The byte-size agreed upon via the rules described for the BYTE-SIZE option. The value of BYTE-SIZE is an integer. For details on the ramifications of BYTE-SIZE: See the section "NFILE OPEN Optional Keyword/Value Pairs", section 8.20.1. This parameter is only meaningful for BINARY openings. However, if FILEPOS is returned in the other-properties list, BYTE-SIZE should also be included, even for character openings.

CREATION-DATE

The creation date of the file. The date is expressed in Universal Time format, which measures a time as the number of seconds since January 1, 1900, at midnight GMT. Creation date does not necessarily mean the time the file system created the directory entry or records of the file. For systems that support modification or appending to files, it is usually the modification date of the file. Creation date can mean the date that the bit count or byte count of the file was set by an application program.

Some types of file systems support a user-settable quantity (CREATION-DATE) which the user can set to an arbitrary time, to indicate that the contents of this file were written a long time ago by someone else on another computer. The default value of this quantity, if the user has not set it, is the time someone last modified the information in the file. This quantity, in the OPEN response for an output file, is disregarded by the user side, but nevertheless must be present.

The Symbolics computer system software uses this quantity as a unique identifier of file contents, for a given file name, type, and version, to prove that a file has not changed since it last recorded this quantity for a file.

FILEPOS

An integer giving the position of the logical file pointer, in characters or bytes as appropriate for the type of opening. This is always zero for an input opening and for an output opening creating a new file. For an output opening appending to an existing file, FILEPOS is the number of characters or bytes, as appropriate, currently in the file. This number, for character openings, is measured in server units: See the section "NFILE Character Set", section 6.

LENGTH

An integer reporting the length of the file, in characters for character openings and in bytes of the agreed-upon size for binary openings. LENGTH should be reported as zero for output openings, even if appending to an existing file. The server usually only knows the length for a character opening in server units; thus, it reports length in server units.

8.21 PROPERTIES Command

Command: (PROPERTIES tid handle pathname control-keywords properties)

Response: (PROPERTIES tid property-element settable-properties)

PROPERTIES requests the property information about one file. The file is identified by the pathname argument or the handle argument, but not both. If pathname is supplied, it is a string in the full pathname syntax of the server host. See the section "Syntax of File and Directory Pathname Arguments", section 7.4.

If handle is supplied, its value is a string identifying an opening, which implicitly identifies a file. For direct access mode openings, handle must be a direct file identifier.

control-keywords is reserved in the current design. However, it is a required argument, and must be supplied as the empty token list. Its presence in the NFILE specification allows for future expansion. In the future the value of control-keywords might affect the listing mode.

properties is a token list of keywords indicating the properties the user wants returned. (In command arguments, properties cannot be specified with integers, such as indices into the Property Index Table). For a list of keywords associated with file properties: See the section "Format of NFILE File Property/Value Pairs", section 7.5.

The server is always free to return more properties than those requested in the properties argument. If properties is supplied as the empty token list, the server transmits all known properties of the file.

PROPERTIES COMMAND RESPONSE

The server returns the property information for the given file in the command response. The PROPERTIES command does not use any data channels. If the specified file does not exist or is not accessible, the server signals an error and includes an appropriate three-letter error code in the command error response. See the section "NFILE Errors and Notifications", section 10.

The return value property-element is a token list. The first element in that token list is the pathname of the file, in the full pathname syntax of the server host. The following elements of the property-element token list are property/value pairs. The server is expected to return several property/value pairs; the number of pairs is not constrained. For further details on file properties and their associated values: See the section "Format of NFILE File Property/Value Pairs", section 7.5.

The return value settable-properties is a token list of keywords. The number of keywords is not constrained. (Note that integers cannot be used in settable-properties to indicate the file property; keywords are to be used instead.) Each keyword supplied in settable-properties identifies a property considered settable by the server. The server is implicitly guaranteeing a mechanism for changing the properties reported as settable. The user can change any of the settable properties for this file by using the CHANGE-PROPERTIES command. See the section "CHANGE-PROPERTIES Command", section 8.2.

The following example shows the format of the PROPERTIES command response. Remember that the number of property/value pairs and keywords is not constrained; this example has two property/value pairs and three settable-properties keywords returned:

```
TOP-LEVEL-LIST-BEGIN
PROPERTIES          - name of the command
tid                 - transaction identifier
LIST-BEGIN
pathname of file
prop1 value1        - file's property/value pairs
prop2 value2
LIST-END
LIST-BEGIN
keyword-1           - file's settable properties
keyword-2
keyword-3
LIST-END
TOP-LEVEL-LIST-END
```

The following example is designed to better show the structure of the top-level token list by depicting TOP-LEVEL-LIST-BEGIN and TOP-LEVEL-LIST-END by parentheses and LIST-BEGIN and LIST-END by square brackets. The indentation and newlines in the example are not part of the token list, but are used here to make the structure of the token list clear.

```
(PROPERTIES tid [ pathname prop1 value1 prop2 value2 ...]
                [ keyword1 keyword2 keyword3 ... ]
```

8.22 READ Command

Command: (READ tid direct-file-id input-handle count FILEPOS)

Response: (READ tid)

READ requests input data flow for direct access openings. The direct-file-id is the same as the DIRECT-FILE-ID argument that was given when opening the file; it designates the opening from which the characters or bytes are to be transferred. The input-handle specifies which data channel should be used for the transfer of data from server to user. The data channel should have been already established, cannot have been disestablished, and must not currently be in use.

count is an integer specifying how many bytes (or NFILE characters, as appropriate) to read. count can be supplied as the empty token list, meaning read to the end of the file. If the user specifies the empty token list or a count greater than the number of bytes remaining in the file, the server sends the keyword EOF to mark the end of the file.

FILEPOS is an optional keyword/value pair. If the keyword FILEPOS is supplied, it must be followed by an integer. Before data is transferred, the opening is positioned to the point specified by the value of FILEPOS. The position of the point is measured in server units for character openings; for binary openings it is measured in binary bytes. See the section "FILEPOS NFILE Command".

Upon receiving the READ command, the server binds the data channel to the opening and immediately begins transferring data. The server stops when all data has been transferred. After the server sends the last requested byte, it unbinds the data channel, freeing it for other use. When the user side has processed the last byte, the user side assumes that the data channel can now be reused for another data transfer.

8.23 RENAME Command

Command: (RENAME tid handle pathname to-pathname)

Response: (RENAME tid from-pathname to-pathname)

RENAME requests the server to give a file a new name. This is NFILE's interface to the system's native rename operation, with all of its system-specific semantics and constraints.

Either a handle or a pathname (but not both) specifies the file that is to receive a new name. The argument to-pathname designates that new name. The return value from-pathname gives the full original name of the file, and to-pathname gives the full new name of the file. For systems that support version numbers, the return values can differ in version number from the values of the arguments given to RENAME.

The arguments pathname and to-pathname and the return values from-pathname and to-pathname are strings in the full pathname syntax of the server host. See the section "Syntax of File and Directory Pathname Arguments", section 7.4.

If the file to be renamed is specified by a pathname, the file should be renamed immediately. If the file is specified by handle, it is acceptable to wait until close-time to rename the file.

Some operating systems can rename only within a directory. Nevertheless, the to-pathname of the RENAME must be fully specified; the server on these systems must check for and reject an attempted cross-directory rename.

8.24 RESYNCHRONIZE-DATA-CHANNEL Command

The command and response format for this command varies, depending on whether the handle argument indicates an input or output data channel.

For an Input Handle:

Command: (RESYNCHRONIZE-DATA-CHANNEL tid handle)

Response: (RESYNCHRONIZE-DATA-CHANNEL tid identifier)

For an Output Handle:

Command: (RESYNCHRONIZE-DATA-CHANNEL tid handle identifier)

Response: (RESYNCHRONIZE-DATA-CHANNEL tid)

RESYNCHRONIZE-DATA-CHANNEL begins a prescribed procedure between user and server over the unsafe data channel specified by handle. The resynchronization procedure clears the data channel of any unwanted data, and restores the data channel to a safe state, ready to transfer data again.

All arguments to RESYNCHRONIZE-DATA-CHANNEL are required.

For a detailed description of how the user and server coordinate the resynchronization of data channels: See the section "NFILE Data Connection Resynchronization", section 9.2.

8.24.1 Implementation Hints for RESYNCHRONIZE-DATA-CHANNEL Command

In general, both the user and server should should be implemented with the knowledge that a transmission can be aborted. That is, the receiving side must be careful not to act upon a transmission (that is, to perform any action or side effect) until the transmission has been successfully received in entirety. This protects the user program from the possibility that an abort can occur after a transmission has been partially sent.

RESYNCHRONIZING AN OUTPUT DATA CHANNEL

The server will probably want to dispatch the looping and reading to the logical data process. Looping reading for the resynchronization identifier in the control connection handler is not a viable option. If the user side fails to send the resynchronization identifier (for example, due to a user abort) the control connection handler can never be broken out of this loop.

Should the user side send the control connection handler command first, or send the marks and identifiers first?

Sending the marks first is problematic, because the data channel at the other end might not be reading them (for it has not yet been so instructed by the control connection handler). The user might then become blocked for output, thus prohibiting sending of the RESYNCHRONIZE-DATA-CHANNEL command.

On the other hand, sending the control connection handler command first requires that the user side can send the marks and identifiers between sending the control connection handler command and receiving a response for it. The response will never come until the marks and identifiers have been successfully received. The user implementation must allow for this one case of a command where a subroutine that "sends a command and waits for a response" is inapplicable.

RESYNCHRONIZING AN INPUT DATA CHANNEL

The server control process should dispatch the data process to send the mark, and not wait, lest the data process become blocked for output due to a user abort. The control process must go back to its command loop, to possibly receive a command that might break the data process out of that block.

8.25 UNDATA-CONNECTION Command

Command: (UNDATA-CONNECTION tid input-handle output-handle)

Response: (UNDATA-CONNECTION tid)

UNDATA-CONNECTION explicitly disestablishes a data connection from the user side. The user side has the option of disestablishing data connections at its discretion. There is no place in the protocol where disestablishment of data connections is required, other than at the end of the session, where it is implicit.

The data connection to be disestablished is the one designated by the input-handle and output-handle arguments. These two handles must refer to the same data connection.

It is not permitted to explicitly disestablish a data connection either of whose channels is active. If the session is terminated by the breaking of the control connection, all file handles become meaningless, and the server must close all data connections known to it and close-abort all files opened on behalf of the user during the dialogue.

In the Symbolics implementation, the user side disestablishes data connections that have not been used for a long time, such as twenty minutes or so.

For more information about data connections: See the section "NFILE Control and Data Connections", section 4.

9. NFILE RESYNCHRONIZATION PROCEDURE

Ordinarily, the user side sends NFILE commands to the server side over the control connection; the server side responds to every user command, and file data is transmitted over the data channels. This section describes a resynchronization procedure that takes place when something disturbs the usual course of events.

First, if the server side aborts while sending or receiving data, nothing can be done to salvage the connection between the two hosts. The control connection and any data channels associated with this connection are broken. This happens rarely, if at all.

It is not unusual for the user side to abort file operations, either commands or data transfer. On a Symbolics computer, the user can do this by pressing CONTROL-ABORT. An important aspect of any file protocol is the way it handles the situation when the user side aborts file operations.

An NFILE user side reacts to user side aborts by immediately marking the connection unsafe. When a control connection is unsafe, it must be resynchronized before it can be used again. Data channels can also be marked unsafe, and must also be resynchronized before further use. The resynchronization process rids the connection (whether control or data connection) of bytes of data that are now unwanted, and thus cleans up the channel so it can be used again.

The resynchronization procedure is somewhat complex, but it fulfills a genuine need. For those interested, a brief design discussion is included as note <3>.

9.1 NFILE Control Connection Resynchronization

NFILE requires any unsafe control connection to undergo a resynchronization procedure before further use. Therefore, the resynchronization does not necessarily occur immediately after the control connection is marked unsafe. The user side initiates the control connection resynchronization when another operation on the control connection is attempted.

A "mark" is defined in the context of Byte Stream with Mark: See the section "Discussion of Byte Stream with Mark", section 12.1.

USER SIDE STEPS: CONTROL CONNECTION RESYNCHRONIZATION

1. The user side sends a mark over the control connection to the server.

2. The user side sends the ASCII characters USER-RESYNC-DUMMY (as a data token) to the server.

3. The user side sends a second mark to the server.

4. The user side declares the control connection safe (at the token list level).

5. The user side generates and sends a unique data token to the server.

6. The user side then waits, expecting to detect a mark followed by the unique data token. The user side reads and discards all tokens and marks until the desired match is found.

Once the user side detects the mark and unique data token, the control connection has been fully resynchronized, and can be used again.

SERVER SIDE STEPS: CONTROL CONNECTION RESYNCHRONIZATION

1. The server side detects a mark. The server is thus alerted that the control connection is unsafe, and that resynchronization is in progress.

2. The server continues to read data coming from the user side until it detects the second mark, and the token following it.

3. The server checks to see if the token following the mark is USER-RESYNC-DUMMY. This rare situation occurs if the user aborts during the course of the resynchronization itself. If so, the server side discards the USER-RESYNC-DUMMY token. The control connection is still unsafe, and the user side restarts the resynchronization procedure; the server side therefore begins at Step 2 again.

4. If the token following the mark is not USER-RESYNC-DUMMY (this is the expected circumstance), the server should have received a single data token that is the unique data token generated by the user side.

    a. The server sends a mark to the user side.

    b. The server declares the control connection safe (at the token list level).

    c. The server sends the unique data token to the user side.

5. If the server detects something following the mark that was neither USER-RESYNC-DUMMY nor a single data token, a protocol error has occurred.

9.2 NFILE Data Connection Resynchronization

The NFILE data channel resynchronization procedure is similar to the NFILE control connection resynchronization. Both procedures are based on a mark signalling the unsafe condition, then a second mark followed by a unique identifier. One important difference between the two procedures is the circumstances in which they occur. Control connections are put into unsafe states only when the user aborts during control connection I/O operations. Data channels are made unsafe by a larger set of circumstances:

- User aborts occur during the file protocol operations that assign and deassign data channels. This is the most common cause of data channels becoming unsafe.

- A server receives a CLOSE command (with abort-p supplied as Boolean truth) specifying an open file that has not finished transmitting data. That is, file reading is aborted.

- The ABORT command is issued, causing data channels to be made unsafe.

- The FILEPOS command is issued, causing the input data channel to become unsafe.

The resynchronization clears the data channel of unwanted data from aborted operations and puts the data channel in a known state. The data channel resynchronization procedure is invoked when the user side gives the RESYNCHRONIZE-DATA-CHANNEL command over the control connection.

The following policies can be used to improve response time, but are not required by the NFILE protocol: The user side can initiate resynchronization only if it needs the data channel, having first tried to use a free data channel that does not require resynchronization. Also, the user side can periodically resynchronize all unsafe data channels.

In giving the RESYNCHRONIZE-DATA-CHANNEL command, the user side indicates which data channel should be resynchronized. Data channels are unidirectional, which means that depending on the direction (either input or output) of the data channel, either the user side or the server side sends the resynchronization data. This is another difference from the resynchronization of the control connection, in which the resynchronization data is always sent by the user side. The resynchronization steps for input data channels are different than the steps for output data channels.

INPUT DATA CHANNEL RESYNCHRONIZATION

1. The user side gives the RESYNCHRONIZE-DATA-CHANNEL command on the control connection, with only one argument, the handle of the data channel to be resynchronized.

2. The server side of the data channel generates a unique identifier, and sends that data token in its regular command response to the user side.

3. The server side sends a mark over the data channel.

4. The server side sends the unique identifier token over the data channel.

5. The user side reads until it detects a mark followed by the unique identifier token. The resynchronization is then complete. The data channel is no longer in an unsafe state.

OUTPUT DATA CHANNEL RESYNCHRONIZATION

1. The user side gives the RESYNCHRONIZE-DATA-CHANNEL command on the control connection, with two arguments: the handle of the data channel to be resynchronized, and a unique identifier that it has just generated.

2. The user side of the data channel sends a mark.

3. The user side of the data channel sends a dummy identifier token. The dummy identifier can be any token that the server could not interpret as being the unique identifier. One suggestion is the data token DUMMY-IDENTIFIER.

4. The server side of the data channel was alerted by the RESYNCHRONIZE-DATA-CHANNEL command that resynchronization is in progress. The server side now reads the data, seeking the first mark.

5. The server side reads and discards the first mark and the dummy identifier.

6. The user side sends a second mark.

7. The user side sends the unique identifier.

8. The server side recognizes the mark and the unique identifier that follows, and the resynchronization is

complete. The data channel is no longer in the unsafe state.

10. NFILE ERRORS AND NOTIFICATIONS

NFILE recognizes two types of errors: command response errors and asynchronous errors. In addition to errors, NFILE supports notifications.

Command response errors:

- Signify an error that prevented the successful completion of the command; when such an error occurs, a command response error is sent instead of a normal command response.
- Occur frequently in normal operations

Asynchronous errors:

- Are not related to any specific command
- Are associated with an erring data channel
- Typically indicate a problem in the transfer, such as running out of disk space or allocation, or an unreadable disk record
- Occur rarely in normal operations

Notifications:

- Are not associated with an error
- Are sent at the server's discretion
- Provide general information, such as a warning that the system is going down

10.1 Notifications From the NFILE Server

The NFILE server can send asynchronous notifications to the user side over the control connection. The text of the notification contains information of interest to the person using NFILE, such as a warning that the server's operating system will be going down soon. Notifications can come from the server side at any time that the server is not sending something else.

The format of NFILE notifications is:

```
(NOTIFICATION "" text)
```

The empty string "" takes the place of a transaction identifier. Notifications are initiated by the server, and are not associated with any transaction originated by the user side.n

10.2 NFILE Command Response Errors

When an error prevents the successful completion of an NFILE command, a command response error is sent instead of the normal command response. A normal command response indicates success; a command response error indicates failure of the command.

NFILE command response errors are sent from the server to the user across the control connection as top-level token lists, in this format:

(ERROR tid three-letter-code error-vars message)

ERROR is a keyword. The tid is the transaction identifier of the command that encountered this error. The arguments three-letter-code, error-vars, and message are all required.

The three-letter-code provides the information on what kind of an error was encountered. For a table of the three-letter codes and their meanings: See the section "NFILE Three-letter Error Codes", section 10.4.

message is a string that is displayed to the human user of the protocol.

error-vars is a keyword/value list. The three possible keywords are: PATHNAME, OPERATION, and NEW-PATHNAME. Before transmitting an error, the server looks at the type of error to see if it can easily determine the value of any of the keywords. If so, the server includes the keyword/value pair in its error. If not, the keyword/value pair is omitted. The value associated with OPERATION is the keyword naming the NFILE command that failed. The values associated with PATHNAME and NEW-PATHNAME are strings in the full pathname syntax of the server host.

For example, suppose the server on a file system with hierarchical directories could not access a file because its containing directory did not exist. The command error response would use the PATHNAME keyword to indicate the first directory level that did not exist, instead of the full pathname which was supplied as the command argument. This gives the user side valuable information that it otherwise would not have known.

10.3 NFILE Asynchronous Errors

When a data channel process, in either direction, encounters an error condition, the server sends an asynchronous error description. An asynchronous error description consists of a top-level token list. Typically, asynchronous errors indicate error conditions in the transfer, such as running out of disk space or allocation, or a unreadable disk record.

The format of asynchronous error descriptions is:

(ASYNC-ERROR handle three-letter-code error-vars message)

ASYNC-ERROR is a keyword. The handle argument identifies the erring data channel. The arguments three-letter-code, error-vars, and message are all required. Their meanings are the same as in NFILE command error responses: See the section "NFILE Command Response Errors", section 10.2.

When the server detects an asynchronous error on an input data channel, the server sends an asynchronous error description on that data channel itself. When an asynchronous error occurs on an output data channel, the asynchronous error description is sent on the control connection.

Some asynchronous errors are restartable. In this context, restartable means it makes sense to try to resume the operation. One example of a restartable error is an attempt to write a file to a file system that is out of room. The server side indicates whether an asynchronous error is restartable by prepending the keyword RESTARTABLE and the associated value Boolean truth to the error-vars list. To proceed from a restartable error, the user side sends a CONTINUE command over the control connection.

On any asynchronous error, either input or output, the data channel on the server side enters an "asynchronous error outstanding" state. The server can exit that state in one of two ways: by receiving a CONTINUE command or a CLOSE command with the abort-p argument supplied as Boolean truth.

On a normal CLOSE (not a close-abort), the server side checks the channel it was requested to close. If an asynchronous error description has been sent on the data channel, but not yet processed by CONTINUE, the server side does not close the channel, but sends a command error response. The same thing happens on a FINISH command received on a channel that has an asynchronous error pending. In both cases, the three-letter code included in the command error response is EPC, for Error Pending on Channel.

10.4 NFILE Three-letter Error Codes

Usually the server's operating system provides some description of an error that occurs. NFILE has a mechanism for conveying that information to the user side. Upon detecting an error, the NFILE server should characterize the error by choosing the three-letter code that best describes the error. The three-letter code is an argument in both the command response error and asynchronous error messages from the server to the user.

Each of the NFILE three-letter codes represents some system error. The set of codes enables all operating systems to use one error-reporting mechanism. Some operating systems will never encounter certain of the error conditions.

Some errors fit logically into two error codes. For example, suppose the server could not delete a file because the file was not found. This error could be considered either CDF (Cannot Delete File) or FNF (File Not Found). In this case, File Not Found gives more specific and valuable information than Cannot Delete File. Since the protocol does not allow more than one error code to be reported when an error occurs, the server must choose the most appropriate error code, given the information available to it from the operating system.

This is the set of three-letter codes:

ACC Access error. This indicates a protection-violation error.

ATD Incorrect access to directory. A directory could not be accessed because the user's access rights to it did not permit this type of access.

ATF Incorrect access to file. A file could not be accessed because the user's access rights to it did not permit this type of access.

BUG File system bug. This includes all protocol violations detected by the server, as well as by the host file system.

CCD Cannot create directory. An error occurred in attempting to create a directory.

CDF Cannot delete file. The file system reported that it cannot delete a file.

CCL Cannot create link. An error occurred in attempting to create a link.

CIR Circular link. An operation was attempted on a pathname that designates a link that eventually links back to itself.

CRF Cannot rename file. An error occurred in attempting to rename a file.

CSP Cannot set property. An error occurred in attempting to change the properties of a file. This could mean that you tried to set a property that only the file system is allowed to set, or a property that is not defined on this type of file system.

DAE Directory already exists. A directory could not be created because a directory or file of this name already exists.

DAT Data error. The file system contains unreadable data. This could mean data errors detected by hardware or inconsistent data inside the file system.

DEV Device not found. The device of the file was not found or does not exist.

DND "Do Not Delete" flag set. An attempt was made to delete a file that is marked by a "Do Not Delete" flag.

DNE Directory not empty. An invalid deletion of a nonempty directory was attempted.

DNF Directory not found. The directory was not found or does not exist. This refers specifically to the containing directory; if you are trying to access a directory, and the actual directory you are trying to access is not found, FNF (for File Not Found) should be indicated instead.

EPC Error pending on channel. The server cannot close the channel in attempting to close or finish the channel.

FAE File already exists. The file could not be created because a file or directory of this name already exists.

FNF File not found. The file was not found in the containing directory. The TOPS-20 and TENEX "no such file type" and "no such file version" errors should also report this condition.

FOO File open for output. Opening a file that was already opened for output was attempted.

FOR Filepos out of range. Setting the file pointer past the

end-of-file position or to a negative position was attempted.

FTB File too big. File is larger than the maximum file size supported by the file system.

HNA Host not available The file server or file system is intentionally denying service to user. This does not mean that the network connection failed; it means that the file system is explicitly not available.

IBS Invalid byte size. The value of the "byte size" option was not valid.

ICO Inconsistent options. Some of the options given in this operation are inconsistent with others.

IOD Invalid operation for directory. The specified operation is invalid for directories, and the given pathname specifies a directory, in directory pathname as file format.

IOL Invalid operation for link. The specified operation is invalid for links, and this pathname is the name of a link.

IP? Invalid password. The specified password was invalid.

IPS Invalid pathname syntax. This includes all invalid pathname syntax errors.

IPV Invalid property value. The new value provided for the property is invalid.

IWC Invalid wildcard. The pathname is not a valid wildcard pathname.

LCK File locked. The file is locked. It cannot be accessed, possibly because it is in use by some other process.

LIP Login problems. A problem was encountered while trying to log in to the file system.

MSC Miscellaneous problems.

NAV Not available. The file or device exists but is not available. Typically, the disk pack is not mounted on a drive, the drive is broken, or the like. Operator intervention is probably required to fix the problem, but retrying the operation is likely to succeed after the problem is solved.

NER Not enough resources. For example, a system limit on the number of open files or network connections has been reached.

NET Network problem. The file server had some sort of trouble trying to create a new data connection, or perform some other network operation, and was unable to do so.

NFS No file system. The file system was not available. For example, this host does not have any file systems, or this host's file system cannot be initialized or accessed for some reason, or the file system simply does not exist.

NLI Not logged in. A file operation was attempted before logging in. Normally the file system interface always logs in before doing any operation, but this problem can occur in certain unusual cases in which logging in has been aborted.

NMR No more room. The file system is out of room. This can mean any of several things:

- The entire file system is full.
- The particular volume involved is full.
- The particular directory involved is full.
- The user's allocated quota has been exceeded.

RAD Rename across directories. The devices or directories of the initial and target pathnames are not the same, but on this file system they are required to be.

REF Rename to existing file. The target name of a rename operation is the name of a file that already exists.

UKC Unknown operation. An unsupported file system operation was attempted, or an unsupported command was attempted.

UKP Unknown property. The property is unknown.

UNK Unknown user. The specified user name is unknown to this host.

UUO Unimplemented option. An option to a command is not implemented.

WKF Wrong kind of file. This includes errors in which an invalid operation for a file, directory, or link was attempted.

WNA Wildcard not allowed.

11. TOKEN LIST TRANSPORT LAYER

PURPOSE: The Token List Transport Layer is a protocol that facilitates the transmission of simple structured data, such as lists.

11.1 Introduction to the Token List Transport Layer

The Token List Transport Layer is a general-purpose protocol. The Token List Transport Layer sends "tokens" through its underlying stream. Each token usually represents a simple quantity, such as a string or integer.

Tokens can be organized into "token lists". Special tokens are provided to denote the starting and ending point of lists. The token list transport layer differentiates between "top-level token lists", which are not contained in other lists, and "embedded token lists", which are contained in other lists. Using lists makes it convenient to send structured records, such as commands and command responses of the client protocol. The top-level token lists provide robustness.

The Token List Transport Layer is a general term that includes two separate but related subjects: the "token list stream" and the "token list data stream". The token list stream is commonly used for applications that can easily organize the information to be transmitted into tokens and lists. The token list data stream is more appropriate for transmitting a large volume of data that cannot easily be structured into tokens and lists, such as file data, which is simply a sequence of characters or bytes.

The following table illustrates the main differences between token list streams and token list data streams:

	Token List Data Stream	Token List Stream
Built on:	token list stream	Byte Stream with Mark
Transmits:	stream data	tokens, token lists
Example of use:	NFILE data channels	NFILE control connection

NFILE uses the the Token List Transport Layer, and provides an excellent example of its usefulness. The NFILE commands and command responses are sent over the control connection in a token list stream. File data is sent across each data channel in a token list data stream.

11.2 Token List Stream

11.2.1 Types of Tokens and Token Lists

All numbers in the token list documentation are represented in decimal notation. Bytes are 8 bits long.

TYPES OF TOKENS

Tokens are of the following types:

1. Atomic tokens.

   Atomic tokens are of the following subtypes:

   - Data tokens. A data token consists of a sequence of bytes with an effectively infinite maximum length. In some contexts a data token represents a string; in other contexts, a data token is other arbitrary data.

     Each data token is preceded in the token list stream by a representation of its length in bytes.

     Data tokens that are under 200 bytes long are preceded by one byte containing their length in bytes. That is, a data token of 34 bytes is preceded by one byte of value 34.

     Data tokens 200 bytes or over are preceded by the byte known as PUNCTUATION-LONG, of value 201. After the 201 comes a four-byte-long number (least significant byte first) containing the length of the data token that follows.

   - Numeric tokens. A sequence of bytes that represent and encode a nonnegative binary integer. The largest valid integer is 2^63 - 1.

     Numeric tokens are either short integers (less than 256) or long integers (greater than or equal to 256). Short integers are preceded by the byte known as PUNCTUATION-SHORT-INTEGER, of value 206.

Long integers are begun by PUNCTUATION-LONG-INTEGER, of value 207. One byte follows, containing the length (in bytes) of the long integer. The integer itself is next, least significant byte first.

- Keyword tokens. A sequence of bytes that represent and encode a named identifier of the implemented protocol. Keyword tokens are used by the client protocol to convey a name; the only significance of a keyword token is in its name.

  Each keyword is preceded by the byte known as PUNCTUATION-KEYWORD, of value 208. The data token following PUNCTUATION-KEYWORD represents the name of the keyword as a string. The characters are in upper-case standard ASCII.

- Boolean truth. A special token that represents the Boolean truth value. This token is known as BOOLEAN-TRUTH, of value 209 <4>.

2. Control tokens.

The token list stream supports four control tokens to delimit token lists, and one padding token.

TOP-LEVEL-LIST-BEGIN	202	This control token appears at the start of each top-level token list.
TOP-LEVEL-LIST-END	203	This control token appears at the end of each top-level token list.
LIST-BEGIN	204	This control token appears at the start of each embedded token list.
LIST-END	205	This control token appears at the end of each embedded token list.
PUNCTUATION-PAD	200	This padding token should be ignored by the token list stream. It can be sent to fill buffers.

TOKEN LISTS

A token list consists of a sequence of atomic tokens or token lists. Token lists are begun and ended by control tokens that delimit the token lists. There are three types of token lists:

1. Top-level token lists.

   Top-level token lists begin with TOP-LEVEL-LIST-BEGIN and end with TOP-LEVEL-LIST-END. Top-level token lists are not contained in other lists.

2. Embedded token lists.

   These token lists occur inside other token lists. They begin with LIST-BEGIN and end with LIST-END.

3. The empty token list.

   This is a special example of the embedded token list. In some contexts, the empty token list represents Boolean falsity. An embedded empty token list is composed of a LIST-BEGIN followed immediately by a LIST-END. A top-level empty token list is composed of TOP-LEVEL-LIST-BEGIN followed immediately by TOP-LEVEL-LIST-END.

11.2.2 Token List Stream Example

This section contains an example of some data that can appear on a token list stream. The example is a top-level token list encoding an NFILE DELETE command.

The DELETE command is composed of the following pieces: a TOP-LEVEL-LIST-BEGIN, the keyword DELETE, a data token containing the transaction identifier, a LIST-BEGIN, a LIST-END, a data token containing a pathname of a file to be deleted, and a TOP-LEVEL-LIST-END. This example uses t105 as the transaction identifier, and /usr/max/temp as the pathname.

All numbers in this section are expressed in decimal notation.

The pieces of the command are displayed here in order:

1. TOP-LEVEL-LIST-BEGIN
2. The keyword token whose name is DELETE
3. The data token containing the characters: t105
4. LIST-BEGIN
5. LIST-END

6. The data token containing the characters: /usr/max/temp
7. TOP-LEVEL-LIST-END

Now, let's translate each piece of the command into the bytes that are transmitted through the token list stream.

1. TOP-LEVEL-LIST-BEGIN

   202 represents TOP-LEVEL-LIST-BEGIN

2. The keyword token whose name is DELETE.

   A keyword token is introduced by PUNCTUATION-KEYWORD, which is represented in the token list stream as the byte 208.

   A data token follows, containing the string "DELETE". A data token under 200 bytes long is introduced by one byte containing its length in bytes. The length of this data token is 6 bytes.

   The data token continues with the standard ASCII character set representation of each character in the string DELETE:

```
208     represents PUNCTUATION-KEYWORD
006     represents the length of this data token
068     represents "D"
069     represents "E"
076     represents "L"
069     represents "E"
084     represents "T"
069     represents "E"
```

3. The data token containing the characters: t105

   This data token is begun by its length in bytes (4), and continues with the NFILE character set representation of each character in the string:

```
004     represents the length of this data token
116     represents "t"
049     represents "1"
048     represents "0"
053     represents "5"
```

4. LIST-BEGIN

   204 represents LIST-BEGIN

5. LIST-END

```
205     represents LIST-END
```

6. The data token containing the characters: /usr/max/temp

```
013     represents length of this data token
047     represents "/"
117     represents "u"
115     represents "s"
114     represents "r"
047     represents "/"
109     represents "m"
097     represents "a"
120     represents "x"
047     represents "/"
116     represents "t"
101     represents "e"
109     represents "m"
112     represents "p"
```

7. TOP-LEVEL-LIST-END

```
203     represents TOP-LEVEL-LIST-END
```

11.2.3 Mapping of Lisp Objects to Token List Stream Representation

The Symbolics interface to the token list stream sends Lisp objects through the underlying Byte Stream with Mark and produces Lisp objects on the other end. Not all Lisp objects can be sent in this way. For example, compound objects other than lists are not handled. An appropriate analogy is the sending and reconstruction of list structure via printed representation. These are the types of objects that can be sent, and their representations:

- Lisp strings are represented as data tokens in the NFILE character set. Only 8-bit strings can be sent <5>.

- Keyword symbols are represented as keyword tokens. Although identifiable and reconstructable as keyword symbols, only their names are sent. Any properties, bindings, and the like are not sent.

- T is represented as BOOLEAN-TRUTH.

- NIL is represented as the empty token list.

- Lists are represented as token lists. Circular lists cannot

be sent. See the footnote related to the ambiguity between

NIL and the empty list: See the section "Types of Tokens and Token Lists", section 11.2.1.

- Integers are represented as numeric tokens. Only nonnegative integers less than 2^63 can be sent.

11.2.4 Aborting and the Token List Stream

A token list stream accrues the benefits of the abort management policy of the Byte Stream with Mark on which it is built. In order to fully realize this benefit, some simple rules must be obeyed by any implementation of the token list stream.

The term "transmission" means either an atomic token or a complete top-level token list. A transmission starts with the control token TOP-LEVEL-BEGIN and ends with TOP-LEVEL-END. The top-level token list can contain embedded token lists.

The interface that writes to the token list stream must be capable of writing the representation of entire transmissions. When this interface is called, it must effectively lock the token list stream, and exclude access by other processes until the entire transmission has been encoded and sent.

If the sending is aborted while the stream is locked, the stream enters an "unsafe" state. Trying to send data while the stream is unsafe signals an error. The application and the token list stream must send a mark to cause resynchronization, and allow the token list stream to be used again. When the reading side encounters this mark, it resynchronizes itself according to whatever client protocol is in use.

Similarly, the interface that reads from the token list stream must be capable of reading entire transmissions. When this interface is called, it must lock the stream, excluding access by other processes until the entire transmission has been read.

If the reading is aborted while the stream is locked, the stream enters an unsafe state. The only exit from this unsafe state is by means of receiving a mark. When the stream is unsafe, the only valid operation that can be performed upon it is "read and discard all tokens until a mark is encountered; read and discard that mark; declare the stream safe again".

Depending on the client protocol, the receipt of a mark might cause the reading side to read for further marks. NFILE implements the resynchronization of token list streams, and serves as a useful example: See the section "NFILE Control Connection Resynchronization", section 9.1.

The Symbolics implementation provides the two mark-handling primitives in this way:

1. Send token (or list) preceded by a mark. When the stream is in the unsafe state (on the output side), this is the only permitted output operation (other than closing).

2. Read through to a mark and read the token (or list) following the mark. When the stream is in the unsafe state (on the input side), this is the only permitted input operation (other than closing).

11.3 Token List Data Stream

The token list data stream is a facility to transmit stream data through a token list stream. The token list data stream imposes the following protocol on the data transmitted:

- Data is sent in the format of loose data tokens, not contained in token lists.

- The keyword token EOF indicates that the end of data has been reached.

- Token lists can be transmitted through the token list data stream.

- No loose tokens other than data tokens or the keyword token EOF can be sent.

- Boundaries between data tokens are not signification. The data is considered to be a continuous stream, with the possible exception of marks.

The token list data stream is most appropriate for sending file data. It is expected (but not required) that its typical mode of use is to send a large number of data tokens, with an occasional token list. The design intent was that token lists would be used by the application program to indicate exceptional situations.

Data tokens, the keyword token EOF, and token lists are defined in

the token list stream documentation: See the section "Types of Tokens and Token Lists", section 11.2.1.

The NFILE file protocol provides a good example of the use of token list data streams. NFILE sends file data through token list data streams; each NFILE data channel is a token list data stream. Errors such as disk errors during the reading of a file are conveyed as token lists through the token list data stream.

12. BYTE STREAM WITH MARK

PURPOSE: Byte Stream with Mark is a simple layer of protocol that guarantees that an out-of-band signal can be transmitted in the case of program interruption. Byte Stream with Mark is designed to provide end-to-end stream consistency in the face of user program aborts.

12.1 Discussion of Byte Stream with Mark

INTRODUCTION

Byte Stream with Mark is a reliable, bidirectional byte stream with one out-of-band (but not out-of-sequence) signal called a "mark". The design of Byte Stream with Mark ensures that the mark is always recognizable on the receiving end. The Byte Stream with Mark is built on an underlying stream, which must support the transmission of 8-bit bytes. Byte Stream with Mark has been implemented to run on TCP and Chaos. Marks are implemented differently on the two protocols.

Marks are used to resynchronize the stream when something has occurred to interrupt normal operations. For example, an application layer sending data over the Byte Stream with Mark can abort in the middle of sending that data. Recovery is handled by sending a mark.

In the context of this document, "aborting" is defined as follows: Aborting the current execution of a program means to halt that execution and to abandon it, never to complete it. The data representing the state of the execution are irrevocably discarded.

EXAMPLE OF USE

Byte Stream with Mark is the layer of protocol underlying NFILE. NFILE uses the marks implemented in Byte Stream with Mark to resynchronize control connections or data channels whose synchronization has been lost. For a description of NFILE's use of marks to resynchronize streams: See the section "NFILE Resynchronization Procedure", section 9.

BYTE STREAM WITH MARK ON CHAOSNET

A mark is recognized on Chaosnet by a packet bearing the opcode 201 (octal). There is no data in a mark packet, so the data portion of the packet is ignored. Byte Stream with Mark transmits all data in packets bearing opcode 200 (octal).

If Byte Stream with Mark is implemented on another (non-Chaos) stream that supports opcode-bearing packets, the recommended implementation is the reservation of an opcode for the mark.

BYTE STREAM WITH MARK ON TCP: RECORD MODE

The purpose of Byte Stream with Mark is to guarantee that marks can always be unambiguously identified. Therefore, for TCP (and for any transport layer that does not implement packets natively) a simple record stream is imposed on the stream. The record boundaries serve only to distinguish where a mark can occur. A record consists of a two-byte byte count, most significant byte first, followed by that many bytes of data. A byte count of zero is recognized as a mark.

Both the sending side and the receiving side must rigorously maintain the integrity of the record boundaries. A writer to the stream must never output a byte count without that number of data bytes following. Similarly, a reader of the stream, after reading a byte count, has effectively contracted to read that many bytes from the encapsulated stream, regardless of whether those bytes are requested by the application layer.

MAINTAINING RECORD INTEGRITY

This subsection deals with maintaining record integrity on non-Chaos networks. Since Chaos implements packets natively, no special care is required to maintain record integrity on the Chaos network.

The design discussed here guarantees record integrity; the underlying stream must guarantee data integrity.

The basic design of Byte Stream with Mark on TCP (and other transport layers that do not implement packets natively) is to preserve record integrity by putting clearly demarcated, byte-counted records in the natural records of the encapsulated stream. Therefore, when the outer stream requests a buffer's worth of file data from the encapsulated stream, it expects to receive a buffer containing one entire, ntegral, record of that stream, complete with byte count.

Because of diverse network implementations on different operating systems, the software that implements the encapsulated stream might

not be able to provide integral record buffers to the Byte Stream with Mark implementation. For example, the writing stream could have written records that are much longer than available buffers on the receiving system. In this case, a request to read from the encapsulated stream returns some buffer or some amount of data representing less than an entire Byte Stream with Mark record. The input subroutine of the Byte Stream with Mark implementation must therefore return a region of this (smaller) buffer, representing less than the full Byte Stream with Mark record. Nevertheless, the Byte Stream with Mark must extract the count of the full Byte Stream with Mark record from the first such buffer of each Byte Stream with Mark record, and maintain and update this count as succeeding component buffers are read.

In this case, if the program reading from the Byte Stream with Mark aborts while reading data, the implementation of Byte Stream with Mark must continue to read through the remaining buffers of the Byte Stream with Mark record that has been subdivided in this fashion.

The user side program will have determined that an abort has occurred, and will request the Byte Stream with Mark to read up to and through the next mark. The Byte Stream with Mark will have processed a fractional record, and must discard the remaining buffers of the record now being read.

12.2 Byte Stream with Mark Abortable States

Byte Stream with Mark is designed to provide end-to-end stream consistency in the face of user program aborts. This section describes user program aborts, and how Byte Stream with Mark handles them. In the context of this document, "aborting" is defined as follows: Aborting the current execution of a program means to halt that execution and to abandon it, never to complete it. The data representing the state of the execution are irrevocably discarded.

USER PROGRAM ABORTS AND I/O STREAMS

Aborting the execution of the code that manipulates I/O streams, in general, poses significant problems. Given that a stream is a static data object, and is intended to be used over and over again, aborting the execution of any routine manipulating a stream can leave it in an inconsistent, unusable state.

Many operating systems solve this problem by manipulating a large subset of streams within the confines of the supervisor or executive program, which is not vulnerable to aborts, short of system or network failure. Nevertheless, the need still exists to implement streams outside of the boundaries of the supervisor. Furthermore,

the Symbolics computer environment has no supervisor or executive program, and is thus vulnerable to aborts everywhere.

BYTE STREAM WITH MARK HANDLING OF USER PROGRAM ABORTS

Byte Stream with Mark is designed to be nearly impervious to the aborting of programs using it. Its design is based on careful analysis of all possible states of the stream, and of the effect of aborts of the programs using the stream in each of these states. This section provides that analysis.

A "transmission" is a collection of user data sent by the application level through the Byte Stream with Mark whose end is well-defined, once its start has been recognized. For instance, the token list stream, when using Byte Stream with Mark, sends token lists. When a TOP-LEVEL-LIST-BEGIN has been sent, the containing transmission is not considered complete until the corresponding TOP-LEVEL-LIST-END is read. See the section "Token List Transport Layer", section 11.

The following cases are possible states of the stream when an abort occurs:

1. Abort occurs when the user program is not manipulating the stream.

   This case presents no problem.

2. Abort occurs after a transmission has been partially sent, at a packet or record boundary.

   This implies that the datum that would indicate the successful complete sending of that transmission has been not yet been sent.

   The Byte Stream with Mark state is consistent, but the application level state is not. The application level must determine that the execution of the code composing and sending its transmission was, in fact, aborted, and initiate resynchronization via marks.

   The receiving side must be careful not to act upon a transmission (that is, to perform any action or side effect) until the transmission has been successfully received in entirety. This protects the user program from the possibility that an abort can occur after a transmission has been partially sent.

3. Abort occurs during the sending or receiving of a record.

   This is the most vulnerable state of the mechanism. This case does not occur on packet-oriented media; it is subsumed by the next case.

   This case is handled by minimizing the extent of this window, and killing the connection when and if the situation is detected. Depending on the operating system involved, this window could be minimized by using interrupt-disabling mechanisms, auxiliary processes or tasks, or some other technique.

   For buffered streams, input and output waiting can be done in consistent states, thus minimizing the amount of time manipulating the actual encapsulated stream. For unbuffered streams, a lot of time can be spent in this window. It is expected that unbuffered streams will be exceedingly uncommon. Nevertheless, the implementation of Byte Stream with Mark must detect this case.

4. Abort occurs during the sending or receiving of fundamental units of the lowest-level underlying stream (packets, buffers, or bytes).

   This case is usually handled by inhibiting interrupts, or other forms of masking, in the code implementing the encapsulated stream, since no waiting is possible at unexpected times.

13. POSSIBLE FUTURE EXTENSIONS

NFILE was designed to be extended as the needs of its clients grow, or as new clients with different needs appear. Currently it meets the needs of the Symbolics Genera 7.0 operating system, although its design is intentionally general. If users of other operating systems identify new features that would be useful, they could be added to NFILE. This section illustrates some areas areas where the design of NFILE intentionally accommodates extensions.

- The NFILE protocol encodes commands and responses as text, rather than using prearranged numbers. This means that new commands and responses can be added without having to obtain a new number from a central registry.

- The Token List Transport Layer provides a general substrate for the value-transmission portion of network protocols. In fact, it has been used at Symbolics for other protocols

besides NFILE. The Token List Transport Layer could conveniently be extended to support transmission of other types of values besides those it currently supports.

- The character set to be used for file transfer could be made negotiable.

- The command character set could be made negotiable. Currently there is no negotiation sequence, but one could be added.

- Greater support for more complex file organizations could be added, such as record files, databases, and so on. This could be an extension to the direct access mode facility.

- Currently, the LOGIN command allows the user side to inform the server which version of NFILE it is running. This feature is included in NFILE so that a server can continue to support older versions of the protocol even after new, extended versions have been implemented. However, the specification is currently somewhat vague as to how the server can make use of the version.

- NFILE is not restricted to using TCP or Chaos as its underlying protocol. NFILE can be built on any byte stream protocol that supports reliable transmission of 8-bit bytes and multiple connections.

In addition to the possible future extensions, we would like to mention a known limitation of NFILE.

Currently NFILE requires multiple connections for a single session. That is, the control connection must be separate from the data connections. If NFILE is to be used over a telephone, this requirement poses an inconvenient restriction. It is possible to implement a multiplexing scheme as a level between NFILE and the communication medium.

APPENDIX A
NORMAL TRANSLATION MODE

NORMAL translation mode guarantees the following:

- A file containing characters in the NFILE character set can be written to any NFILE server and read back intact (containing the same characters).

- A file written by NFILE should not appear as "foreign" to a server operating system unless the file contains NFILE's extended characters. That is, a server file that uses only the subset of the NFILE character set limited to standard ASCII characters (the 95 printing characters, and the native representation of return, linefeed, page, backspace, rubout, and tab) can be read and written, with the result being the same data in NFILE characters as exists in server characters.

In this section, all numbers designating values of character codes are to be interpreted in octal. The notation "x in c1..c2" means "for all character codes x such that c1 <= x <= c2."

The NFILE character set is an extension of standard ASCII. The 95 ASCII printing characters have the same numerical codes in the NFILE character set. Five ASCII non-printing characters have counterparts in the NFILE character set, as shown in the following table. The NFILE character set includes a single Return character, rather than the carriage-return line-feed sequence typically used in ASCII. The NFILE character set does not include the ASCII control characters, other than the five shown in the following table, but does include some additional printing and formatting characters that have no counterparts in ASCII.

	NFILE	Standard ASCII
Rubout:	207	177
Backspace:	210	10
Tab:	211	11
Linefeed:	212	12
Page:	214	14

Note that the NFILE Return character is of code 215. This character includes "going to the next line". This is a notable difference from the convention used in PDP-10 ASCII in which lines are ended by a pair of characters, "carriage return" and "line feed".

NORMAL TRANSLATION TO UNIX SERVERS

The translation given in this table is appropriate for use by UNIX servers, or other servers that use 8-bit bytes to store ASCII characters. Machines with 8-bit bytes usually place the extra NFILE characters in the top half of their character set.

TABLE 1. TRANSLATIONS FROM NFILE CHARACTERS TO UNIX CHARACTERS

NFILE character	UNIX character
x in 000..007	x
x in 010..015	x + 200
x in 016..176	x
177	377
x in 200..207	x
x in 210..211	x - 200
212	015
x in 213..214	x - 200
215	012
x in 216..376	x
377	177

TABLE 2. TRANSLATIONS FROM UNIX CHARACTERS TO NFILE CHARACTERS

UNIX character	NFILE character
x in 000..007	x
x in 010..011	x + 200
012	215
x in 013..014	x + 200
015	212
x in 016..176	x
177	377
x in 200..207	x
x in 210..215	x - 200
x in 216..376	x
377	177

NORMAL TRANSLATION TO PDP-10 FAMILY SERVERS

The translation given in this table is appropriate for use by PDP-10 family servers, or other servers that use 7-bit bytes to store ASCII characters. On the PDP-10 the sequence CRLF, 015 012, represents a new line.

The mechanism for this translation on machines with 7-bit bytes is to use the RUBOUT character (octal code 177) as an escape character.

TABLE 3. TRANSLATIONS FROM NFILE TO PDP-10 CHARACTERS

NFILE character	PDP-10 character(s)
x in 000..007	x
x in 010..012	177 x
013	013
x in 014..015	177 x
x in 016..176	x
177	177 177
x in 200..207	177 x - 200
x in 210..212	x - 200
213	177 013
214	014
215	015 012
x in 216..376	177 x - 200
377	no corresponding code

These tables might seem confusing at first, but there are some general rules about it that should make it clearer. First, NFILE characters in the range 000..177 are generally represented as themselves, and x in 200..377 is generally represented as 177 followed by x - 200. That is, 177 is used to quote the second 200 NFILE characters. It was deemed that 177 is a more useful and common character than 377, so 177 177 means 177, and there is no way to describe 377 with PDP-10 ASCII characters. In the NFILE character set, the formatting control characters appear offset up by 200 with respect to standard ASCII. This explains why the preferred mode of expressing 210 (backspace) is 010, and 010 turns into 177 010. The same reasoning applies to 211 (Tab), 212 (Linefeed), 214 (Formfeed), and 215 (Return).

More special care is needed for the Return character, which is the mapping of the system-dependent representation of "the start of a new line". The NFILE Return (215) is equivalent to 015 012 (CRLF) in some ASCII systems. In the NFILE character set there is no representation

TABLE 4. TRANSLATIONS FROM PDP-10 CHARACTERS TO NFILE CHARACTERS

PDP-10 character	NFILE character
x in 000..007	x
x in 010..012	x + 200
013	013
014	214
015 012	215
015 not-012	115
x in 016..176	x
177 x in 000..007	x + 200
177 x in 010..012	x
177 013	213
177 x in 014..015	x
177 x in 016..176	x + 200
177 177	177

of a carriage that doesn't go to a new line, so if there is one in a server file, it must be translated to something else. When converting ASCII characters to NFILE characters, an 015 followed by an 012 therefore turns into a 215. A stray CR is arbitrarily translated into a single M (115).

APPENDIX B
RAW TRANSLATION MODE

RAW mode means no translation should be performed. In RAW mode the server operating system should treat the file as a character file and use the same data formatting that would be appropriate for a character file, but transfer the actual binary values of the character codes.

APPENDIX C
SUPER-IMAGE TRANSLATION MODE

SUPER-IMAGE mode is intended for use by PDP-10 family machines only. It is included largely as an illustration of a system-dependent extension. A server machine that has 8-bit bytes should treat SUPER-IMAGE mode the same as NORMAL mode.

In this section, all numbers designating values of character codes are to be interpreted in octal. The notation "x in c1..c2" means "for all character codes x such that c1 <= x <= c2."

SUPER-IMAGE mode suppresses the use of the 177 character as an escape character. Character translation should be done as in NORMAL mode, with one exception. When a two-character sequence beginning with 177 is detected, the 177 should not be output at all.

In this section, all numbers designating values of character codes are to be interpreted in octal. SUPER-IMAGE mode is intended for use by PDP-10 machines only.

SUPER-IMAGE suppresses the use of Rubout for quoting. That is, for each entry beginning with a 177 in the PDP-10 character column in the NORMAL translation table, the NFILE character has the 177 removed.

TABLE 5. SUPER-IMAGE TRANSLATION FROM NFILE TO ASCII

NFILE character	PDP-10 character(s)
x in 000..177	x
x in 200..214	<x - 200>
215	015 012
x in 216..376	<x - 200>
377	no corresponding code

TABLE 6. SUPER-IMAGE TRANSLATION FROM ASCII TO NFILE

PDP-10 character	NFILE character
x in 000..007	x
x in 010..012	x + 200
013	013
014	214
015 012	215
015 not-012	115
x in <016..176>	x
177	177

NOTES

1. NFILE's requirement for using the NFILE character set is recognized as a drawback for non-Symbolics machines. A useful extension to NFILE would be a provision to make the character set negotiable.

2. Implementation note: Care must be taken that the freeing is done before the control connection is allowed to process another command, or else the control connection may find the data channel to be falsely indicated as being in use.

3. The Symbolics operating system has the policy that whenever the user side is waiting for the server side, a user abort can occur. This user side waiting can occur in any context, such awaiting a response, waiting in the middle of reading network input, or waiting in the middle of transmitting network output. Thus there are no "hung" states.

4. Note that the Token List Transport Layer supplies a special token to indicate Boolean truth, but no corresponding token to indicate Boolean falsity. NFILE uses an empty token list to indicate Boolean falsity. The historical reason for this asymmetry is the inability of the Lisp language to differentiate between the empty list and NIL, which is traditionally used to mean Boolean falsity. If the flexibility of both a Boolean falsity and an empty token list were allowed, it would create problems for an operating system that cannot distinguish between the two. This aspect of the protocol is recognized as a concession to the Lisp language. The unfortunate effect is to disallow operating systems to distinguish between Boolean falsity and an empty list.

5. No so-called "fat strings" can be sent.